# Progress Report - 9 Months

Meshal Binnasban

October 14, 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 regular expression matcher based on marks with the goal of POSIX value extraction. The report first reviews regular expression matching using derivatives, then reviews existing regular expression matchers based on marks, which currently only provide matching functionality but do not support POSIX value extraction, which we are especially interested in. Our work developed several versions of the marked algorithm in Scala. we also started formally proving the correctness of one of our versions.

# 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 [8] extended derivatives with bitcodes to record lexing information, but the size problem persists. The marked approach, described in the works of Fischer et al. [3] 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 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 . . . . . 5
2.1.1 Size Explosion. . . . . 7
2.1.2 Derivative Extension . . . . . 11
2.2 Marked Approach . . . . . 14
2.2.1 Motivation for a Marked Approach . . . . . 15
2.2.2 Scala Implementation of Fischer's Marked Approach . 16

**3 Our Approach** **19**
3.1 Bit-Annotated, version 1 . . . . . 20
3.2 Bit-Annotated, version 2 . . . . . 25
3.3 String-Carrying Marks - Matcher . . . . . 25
3.4 String-Carrying Marks - Lexer **so far** . . . . . 31

**4 Future Work** **31**

**References** **33**

# 1 Introduction

The notion of derivatives in regular expressions is well established but has gained renewed attention in the last decade, for example in the work of Owens [7] and Might [4]. Their simplicity and compatibility with functional programming have encouraged further studies, for example the work of Sulzmann and Lu [8]. However, derivatives suffer from "growth" issues, since each step of taking the derivative can increase the size of subexpressions. This means the algorithm has to traverse larger and larger regular expressions, which results in a slower algorithm.

In contrast, the approach based on marks leaves regular expressions unchanged during matching but moves annotations through them (the marks). At present, only matcher-based algorithms using this approach exist,while our work aims to extend them to provide POSIX value extraction.

This report first reviews Brzozowski's derivatives and the bitcoded variant of Sulzmann and Lu [8], followed by background on the regular expression matchers based on marks described by Fischer et al. [3] and Asperti et al. [2]. We then present our own work, including several versions of the matcher based on marks developed during the first nine months.

# 2 Background

Regular expressions are a way to describe languages of words over an alphabet. They provide a declarative way to specify sets of words. Each regular expression $r$ is associated with a language $L(r)$, which is the set of words that match $r$. The regular expressions we consider are the standard regular expression constructors, including bounded repetitions, namely:

$$r ::= \mathbf{0} \mid \mathbf{1} \mid c \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^* \mid r^{\{n\}}$$

We will use $\mathbf{0}$ for the regular expression which cannot match any string, $\mathbf{1}$ for the regular expression which can match the empty string, and $c$ for the regular expression which can match a character from the alphabet. The operator $+$ is for alternation between $r_1$ and $r_2$, while the dot $\cdot$ is for concatenation or sequencing. The Kleene star $r^*$ is for arbitrary repetition of $r$, and the power $r^n$ is for exactly $n$ repetitions of $r$. The meaning of these regular expressions is given by the language function $L$ (Figure 1). For example, if

$$
\begin{array}{lcl}
L(\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
L(\mathbf{1}) & \stackrel{\text{def}}{=} & \{[\,]\} \\
L(c) & \stackrel{\text{def}}{=} & \{[c]\} \\
L(r_1 + r_2) & \stackrel{\text{def}}{=} & L(r_1) \cup L(r_2) \\
L(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & L(r_1) \cdot L(r_2) \\
L(r^*) & \stackrel{\text{def}}{=} & (L(r))^* \\
L(r^n) & \stackrel{\text{def}}{=} & (L(r))^n
\end{array}
$$

Figure 1: the function $L$ which gives the meaning to the regular expression, i.e. associated set of strings to the language of the regular expression

$r = a + b$, then

$$L(r) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}.$$

Similarly, $L(a \cdot b) = \{ab\}$, and $L(a^*)$ is the set of all strings consisting of zero or more $a$'s.

## 2.1 Derivatives

Brzozowski's derivatives offer an elegant way for string matching (Figure 2). What is beautiful about derivatives is that they can be easily implemented in a functional programming language and are easy to reason about in theorem provers. By successively taking the derivative of a regular expression with respect to each input character, one obtains a sequence of derivatives. If the final derivative can match the empty string, then the original input is accepted. For example, To decide whether a string $a_1 a_2 \ldots a_n$ is matched by a regular expression $r$, we successively compute derivatives:

$$r_0 = r, \quad r_1 = der_{a_1}(r_0), \quad r_2 = der_{a_2}(r_1) \ldots, r_n = der_{a_n}(r_{n-1}).$$

The string matches $r$ if and only if the final expression $r_n$ accepts the empty string, which can be easily checked by a separate function. Here $der_a(r)$ stands for the derivative of $r$ with respect to the character $a$.

$$
\begin{array}{lcl}
der_a(\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
der_a(\mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
der_a(c) & \stackrel{\text{def}}{=} & \begin{cases} \mathbf{1} & \text{if } a = c \\ \mathbf{0} & \text{if } a \neq c \end{cases} \\
der_a(r_1 + r_2) & \stackrel{\text{def}}{=} & der_a(r_1) + der_a(r_2) \\
der_a(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & der_a(r_1) \cdot r_2 \; + \; \begin{cases} der_a(r_2) & \text{if } \varepsilon \in L(r_1) \\ \mathbf{0} & \text{otherwise} \end{cases} \\
der_a(r^*) & \stackrel{\text{def}}{=} & der_a(r) \cdot r^*
\end{array}
$$

Figure 2: Brzozowski's Derivative

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 with respect to a (see Figure 3). Since the final derivative expression can match the empty string with no input characters remaining, this confirms that the string `ba` belongs to the language of the regular expression.

Note however,some subexpressions that arise during the computation of derivatives may be redundant. For example, $\mathbf{0} \cdot b$ cannot match any string, so we colud simplify it to $\mathbf{0}$ which can match the same strings, namely none. Other simplification rules may also be applied. 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 $\mathbf{0}$ and $\mathbf{1}$ can also be applied; e.g. $\mathbf{0} \cdot r \equiv r \cdot \mathbf{0} \equiv \mathbf{0}$, $r + \mathbf{0} \equiv r$, and $r \cdot \mathbf{1} \equiv \mathbf{1} \cdot r \equiv r$.

These simplifications preserve the language accepted by the regular expression while reducing overall size of the regular expression. Such simplifications are sometimes necessary in the derivative method in order to keep the size manageable but eventhough best effort still means that the size can explode.

$$
\begin{array}{rcl}
der_b\, r & = & der_b\,(ab + ba) \\
 & = & der_b\,(ab) + der_b\,(ba) \\
 & = & (der_b\, a) \cdot b + (der_b\, b) \cdot a \\
 & = & \mathbf{0} \cdot b + \mathbf{1} \cdot a \\
 & & \\
der_a\,(der_b\, r) & = & der_a\,(\mathbf{0} \cdot b + \mathbf{1} \cdot a) \\
 & = & der_a\,(\mathbf{0} \cdot b) + der_a\,(\mathbf{1} \cdot a) \\
 & = & der_a\,(\mathbf{0}) \cdot b + (der_a\,(\mathbf{1}) \cdot a + der_a\, a) \\
 & = & \mathbf{0} \cdot b + (\mathbf{0} \cdot a + \mathbf{1})
\end{array}
$$

Figure 3: Derivatives of $(ab + ba)$ with respect to string `ba`

### 2.1.1 Size Explosion.

Although the construction of derivatives is elegant, it may duplicate large parts of the expression as well as the sizes of the regular expressions in the intermediate steps (see Figure 2; the sequence case can create additional subexpressions, as can the star case). Sulzmann and Lu [8] also note this problem, describing it as the well-known issue that the size and number of derivatives may explode. The size of derivatives can become very large even for simple expressions. This happens because each derivative is usually more complicated than the original expression, and even when standard simplifications are applied, they may not simplify all duplications. Consider the case of the regular expression $(a + aa)^*$.

a+aa

Each derivative step may introduce new structure, unfolding all possible ways in which the $r^*$ expression can match the input. Even with basic simplifications, many of the resulting subexpressions still describe the same language but in different syntactic forms, and so the size continues to grow.

It is already known that simplification reduces the number of generated expressions and helps to provide a finite bound on the number of intermediate expressions [8, 9], but it does not completely solve the underlying problem. Even with aggressive simplifications—as in Sulzmann and Lu's bitcoded ap-

proach and in the variant described by Tan and Urban—the resulting derivatives can still be large even if finitely bounded. For example, expressions of the form

$$((a)^* + (aa)^* + (aaa)^* + \dots)^*$$

cannot be simplified completely, since the duplicates occur in different levels of the dertivatives and different levels of sequences and alternatives. Thus, even under aggressive simplification the size of derivatives remains large and continues to grow across derivative steps. If we consider the expression

$$((a)^* + (aa)^* + (aaa)^* + (aaaa)^* + (aaaaa)^*)^*$$

with a small input string of the form $\underbrace{a \dots a}_{n}$, each derivative step introduces subexpressions that are copies of the original, as shown in Figure 4.

Figure 5 illustrates the size explosion issue of derivatives by showing the runtime difference between derivatives with aggressive simplifications and Fischer's mark-based algorithm. We conducted a naive test using an example of the form of regular expressions mentioned just above to demonstrate the impact of the growth of derivatives, even under aggressive simplifications. Although the test is naive, it clearly shows a considerable difference in behaviour. The test was designed for inputs of up to 100,000 characters; however, the derivatives implementation ran out of memory after 10,000, with the last successful case taking almost 22 seconds, while the mark-based algorithm finished executing all inputs with each case under 0.0035 seconds.

Even Antimirov's partial derivatives [1], which do not track POSIX values, may still produce cubic growth in the worst case: there can be linearly many distinct derivatives, and each one may already be quadratic in the size of the original expression, so the total size becomes cubic.

$$
\begin{array}{rcl}
der_a\, r & = & der_a\, ((a)^* + (aa)^* + (aaa)^* + (aaaa)^* + (aaaaa)^*)^*) \\
 & = & der_a\, ((a)^* + (aa)^* + (aaa)^* + (aaaa)^* + (aaaaa)^*) \cdot r \\
 & = & ((der_a\, (a)^*) + (der_a\, (aa)^*) + \ldots) \cdot r \\
 & = & ((der_a\, a \cdot (a)^*) + (der_a\, aa \cdot (aa)^*) + \ldots) \cdot r \\
 & = & ((\mathbf{1} \cdot (a)^*) + ((\mathbf{1} \cdot a) \cdot (aa)^*) + \ldots) \cdot r \\
 & = & ((a)^* + (a \cdot (aa)^*) + \ldots) \cdot r \\
 & & \\
der_a\, (der_a\, r) & = & der_a\, ((a)^* + (a \cdot (aa)^*) + \ldots) \cdot r + der_a\, r \\
 & = & (\mathbf{1} \cdot (a)^* + (\mathbf{1} \cdot (aa)^*) + \ldots) \cdot r + \\
 & & ((a)^* + (a \cdot (aa)^*) + \ldots) \cdot r \\
 & = & ((a)^* + ((aa)^*) + \ldots) \cdot r + \\
 & & ((a)^* + (a \cdot (aa)^*) + \ldots) \cdot r \\
 & & \\
der_a\, (der_a\, (der_a\, r)) & = & der_a\, (((a)^* + ((aa)^*) + \ldots) \cdot r) + \\
 & & der_a\, (((a)^* + (a \cdot (aa)^*) + \ldots) \cdot r) \\
 & = & (der_a\, (((a)^* + ((aa)^*) + \ldots) \cdot r) + der_a\, r) + \\
 & & (der_a\, (((a)^* + (a \cdot (aa)^*) + \ldots) \cdot r) + der_a\, r)
\end{array}
$$

Figure 4: Derivatives of $((a)^* + (aa)^* + (aaa)^* + (aaaa)^* + (aaaaa)^*)^*$ with respect to string `aaa`

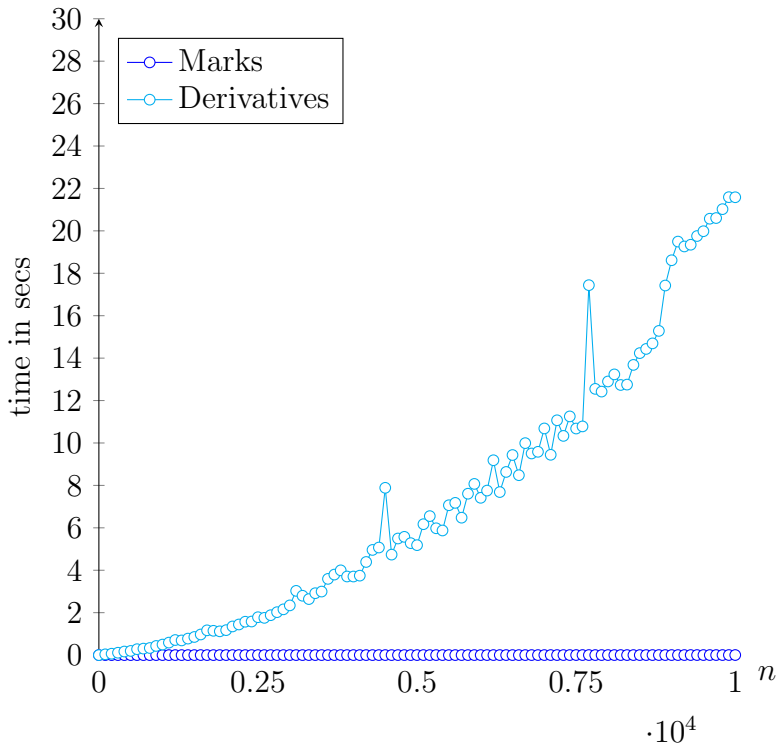


Figure 5: Runtime comparison of derivative-based and mark-based regular expression matchers for $((a)^* + (aa)^* + (aaa)^* + (aaaa)^* + \ldots)^*$ with input strings of length $n$.

### 2.1.2 Derivative Extension

Sulzmann and Lu [8] extend Brzozowski's derivatives to produce lexing values in addition to deciding whether a match exists. These values record how the match occurred, for example by showing which part of the regular expression matched the input string, whether the left or right branch was taken, and how sequences and Kleene stars were matched.

Sulzmann and Lu provide two variants: a bitcode-based construction and an injection-based construction [8]. In the bitcode variant, bit sequences encode lexing choices during derivative construction and, after acceptance, are decoded to the value. In the injection variant, an *inj* function "injects back" the consumed characters into the value; the injection function essentially reverts the derivative steps to obtain a POSIX value. We focus here 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 $(a + ab)(b + \mathbf{1})$ and the string `ab` (see Figure 6). 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 [8]. 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 $a$.

The result of Step 2 shows that the $\mathbf{1}$ 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 *mkeps*, as defined by Sulzmann and Lu [8], 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

1. Step 1: Internalizing the regular expression

$$r' \;=\; intern(r) = ({}_0a + {}_1ab) \cdot ({}_0b + {}_1\mathbf{1}\,)$$

2. Step 2: input a

$$\begin{array}{rcl} der_a(r') & = & der_a(({}_0a + {}_1ab) \cdot ({}_0b + {}_1\mathbf{1}\,)) \\ & = & der_a({}_0a + {}_1ab) \cdot ({}_0b + {}_1\mathbf{1}\,) \\ & = & (der_a({}_0a) + der_a({}_1ab)) \cdot ({}_0b + {}_1\mathbf{1}\,) \\ & = & ({}_0\mathbf{1} + der_a({}_1a) \cdot b) \cdot ({}_0b + {}_1\mathbf{1}\,) \\ & = & ({}_0\mathbf{1} + {}_1\mathbf{1} \cdot b) \cdot ({}_0b + {}_1\mathbf{1}\,) \end{array}$$

3. Step 3: input b

$$\begin{array}{rcl} der_b(der_a(r')) & = & der_b(({}_0\mathbf{1} + {}_1\mathbf{1} \cdot b) \cdot ({}_0b + {}_1\mathbf{1}\,)) \\ & = & der_b({}_0\mathbf{1} + {}_1\mathbf{1} \cdot b) \cdot ({}_0b + {}_1\mathbf{1}\,) \\ & & + der_b(fuse(mkeps(r_1), ({}_0b + {}_1\mathbf{1}\,))) \\ & = & (der_b({}_0\mathbf{1}\,) + der_b({}_1\mathbf{1} \cdot b)) \cdot ({}_0b + {}_1\mathbf{1}\,) \\ & & + {}_0(der_b({}_0b) + der_b({}_1\mathbf{1}\,)) \\ & = & (\mathbf{0} + der_b({}_1\mathbf{1} \cdot b)) \cdot ({}_0b + {}_1\mathbf{1}\,) + {}_0({}_0\mathbf{1} + \mathbf{0}\,) \\ & = & (der_b({}_1\mathbf{1}\,) \cdot b + der_b({}_1b)) \cdot ({}_0b + {}_1\mathbf{1}\,) + {}_0({}_0\mathbf{1} + \mathbf{0}\,) \\ & = & (\mathbf{0} \cdot b + {}_1\mathbf{1}\,) \cdot ({}_0b + {}_1\mathbf{1}\,) + {}_0({}_0\mathbf{1} + \mathbf{0}\,) \end{array}$$

---

Figure 6: Bitcoded derivatives of $(a + ab) \cdot (b + \mathbf{1})$ with respect to string `ba`

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 [8] use $fuse(mkeps(r_1))$ to include the bit annotations needed when $r_1$ is skipped; these bits, extracted by *mkeps*, indicate how $r_1$ matched the empty string.

After taking the derivatives with respect to the entire input string, the algorithm checks whether the result is nullable. If so, it calls *mkeps* 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, *mkeps* returns the bitcode $[1, 1]$, which encodes the choice

of the right branch in the alternative of $r_1$ (matching the string $ab$), followed by the right branch in the alternative of $r_2$ (matching the empty string). Decoding this against the original expression yields the POSIX value: $Seq(\, Right(\, Seq(\, a, b\,)), Right(\, Empty\,))$

The formal definitions of the auxiliary functions *intern*, *fuse*, and *mkeps* appear in Figures 7, 8, and 9, as defined by Sulzmann and Lu [8].

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

Figure 7: *intern* function

$$
\begin{array}{lcl}
fuse\ bs\ (c\,{}_{bs'}) & \stackrel{\text{def}}{=} & c\,{}_{(bs \cup bs')} \\
fuse\ bs\ ((r_1 + r_2)\,{}_{bs'}) & \stackrel{\text{def}}{=} & (r_1 + r_2)\,{}_{(bs \cup bs')} \\
fuse\ bs\ ((r_1 \cdot r_2)\,{}_{bs'}) & \stackrel{\text{def}}{=} & (r_1 \cdot r_2)\,{}_{(bs \cup bs')} \\
fuse\ bs\ (r^*\,{}_{bs'}) & \stackrel{\text{def}}{=} & (r^*)\,{}_{(bs \cup bs')} \\
fuse\ bs\ (r^n\,{}_{bs'}) & \stackrel{\text{def}}{=} & (r^n)\,{}_{(bs \cup bs')}
\end{array}
$$

Figure 8: *fuse* function

$$
\begin{array}{lcl}
mkeps(\mathbf{1}_{\,bs}) & \stackrel{\text{def}}{=} & bs \\
mkeps((r_1 + r_2)_{\,bs}) & \stackrel{\text{def}}{=} & \begin{cases} bs \cup mkeps(r_1) & \text{if } nullable(r_1) \\ bs \cup mkeps(r_2) & \text{otherwise} \end{cases} \\
mkeps((r_1 \cdot r_2)_{\,bs}) & \stackrel{\text{def}}{=} & bs \cup mkeps(r_1) \cup mkeps(r_2) \\
mkeps((r^*)_{\,bs}) & \stackrel{\text{def}}{=} & bs \cup [1]
\end{array}
$$

Figure 9: *mkeps* function

## 2.2 Marked Approach

The marked approach is a method for regular expression matching that tracks matching progress by inserting and moving marks into the expression. As noted by Nipkow and Traytel [5], the idea can be traced to earlier work, in particular to Yamada and Glushkov, who identify positions in regular expressions by marking their atoms. Nipkow and Traytel cite Fischer et al. [3] and Asperti et al. [2] as reviving this work and developing it in a modern setting.

This approach allows for more efficient matching, particularly in complex expressions such as $(a^*b^*)^*$, in large alternations like $(a + b + c + \cdots + z)^*$, or in nested choices such as $((a)^* + (aa)^* + (aaa)^* + (aaaa)^* + (aaaaa)^*)^*$. The regular expression itself does not increase in size; rather, progress is tracked through the placement of marks. With each input character, these marks traverse the regular expression 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. [3] 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 matching problem fails.

Nipkow and Traytel, in their work on regular expression equivalence, build a unified framework for decision procedures on regular expressions. They describe McNaughton–Yamada–Glushkov and Fischer's work as mark-after-atom, meaning that when a character is read, the mark is placed immediately after it. In their framework, this is expressed by two operations on marks, namely $follow$ and $read$. The function $follow$ is, as they describe, similar to an epsilon-closure, in that it moves all marks in a regular expression to the next atom to be read, then $read$ marks all characters corresponding to its input argument. The main transition function is therefore written as

$$shift\ m\ r\ x = read\ x\ (follow\ m\ r).$$

By contrast, they show that the works of Asperti et al., although presented as McNaughton–Yamada, are in fact a dual construction, mark-before-atom. In this version the marks indicate atoms that are about to be read, rather than atoms that have just been consumed. The main transition function is the inversion of the previous transition function:

$$move\ c\ r\ m = follow\ m\ (read\ c\ r).$$

### 2.2.1 Motivation for a Marked Approach

The marked approach offers an alternative to derivatives for regular expression matching. It relies on propagating marks 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. [3, 2], we aim to extend the marked approach to extract POSIX values, as well as to handle extended constructors, such as bounded repetitions and intersections. Our work is primarily based on the algorithm described by Fischer et al. [3]. As shown by Nipkow and Traytel [5], 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. [3]. 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.

$$\mathit{shift}(m, c, r) = \begin{cases} \bullet\, r & \text{if } c \text{ matches in } r, \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

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. Shifting marks for the base cases $\mathbf{0}$ and $\mathbf{1}$ is straightforward: $\mathbf{0}$ cannot be marked, and $\mathbf{1}$ —for now—will not carry a mark, since it matches only the empty string. In the initial algorithm, $\mathbf{1}$ was not marked, and this choice is carried over into later versions. The reason is to avoid complications and ensure termination of the *shift* func-

$$
\begin{array}{lcl}
\mathit{shift}(m, c, \mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathit{shift}(m, c, \mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{1} \\
\mathit{shift}(m, c, d) & \stackrel{\text{def}}{=} & \begin{cases} \bullet d & \text{if } c = d \wedge m \\ d & \text{otherwise} \end{cases} \\
\mathit{shift}(m, c, r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{shift}(m, c, r_1) + \mathit{shift}(m, c, r_2) \\
\mathit{shift}(m, c, r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{c}{\begin{cases} \mathit{shift}(m, c, r_1) \cdot \mathit{shift}(\mathit{true}, c, r_2) & \text{if } m \wedge \mathit{nullable}(r_1) \\ \mathit{shift}(m, c, r_1) \cdot \mathit{shift}(\mathit{true}, c, r_2) & \text{if } \mathit{fin}(r_1) \\ \mathit{shift}(m, c, r_1) \cdot \mathit{shift}(\mathit{false}, c, r_2) & \text{otherwise} \end{cases}} \\
\mathit{shift}(m, c, r^*) & \stackrel{\text{def}}{=} & \begin{cases} \mathit{shift}(\mathit{true}, c, r^*) & \text{if } \mathit{fin}(r) \\ \mathit{shift}(m, c, r^*) & \end{cases}
\end{array}
$$

Figure 10: *shift* function in the Scala implementation of Fischer's Marked Approach.

tion, 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

[1] Our choice follows Fischer et al. [3], where **1** 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.

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 $\mathit{fin}(r_1)$ 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 $\mathit{fin}(r)$ holds.

The formal definitions of the auxiliary functions *fin* and *nullable* appear in Figures 11 and 12, as defined by Fischer et al. [3].

$$
\begin{array}{lcl}
\mathit{fin}(\mathbf{0}) & \stackrel{\text{def}}{=} & \text{false} \\
\mathit{fin}(\mathbf{1}) & \stackrel{\text{def}}{=} & \text{false} \\
\mathit{fin}(c) & \stackrel{\text{def}}{=} & \text{false} \\
\mathit{fin}(\bullet\, c) & \stackrel{\text{def}}{=} & \text{true} \\
\mathit{fin}(r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{fin}(r_1) \vee \mathit{fin}(r_2) \\
\mathit{fin}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & (\mathit{fin}(r_1) \wedge \mathit{nullable}(r_2)) \vee \mathit{fin}(r_2) \\
\mathit{fin}(r^*) & \stackrel{\text{def}}{=} & \mathit{fin}(r)
\end{array}
$$

Figure 11: *fin* function

$$
\begin{array}{lcl}
\mathit{nullable}(\mathbf{0}) & \stackrel{\text{def}}{=} & \text{false} \\
\mathit{nullable}(\mathbf{1}) & \stackrel{\text{def}}{=} & \text{true} \\
\mathit{nullable}(c) & \stackrel{\text{def}}{=} & \text{false} \\
\mathit{nullable}(r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}(r_1) \vee \mathit{nullable}(r_2) \\
\mathit{nullable}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2) \\
\mathit{nullable}(r^*) & \stackrel{\text{def}}{=} & \text{true}
\end{array}
$$

Figure 12: *nullable* function

# 3 Our Approach

We began our work by implementing the marked approach described by Fischer et al. [3] in Scala. This initial implementation of the algorithm included 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 values. This happens in cases where the marks are 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 always produce the POSIX value. Initially, we modified the previous version to accumulate all possible paths to a match, retaining every bitsequence 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 value. One resulting difficulty was the proliferation of matches in the *Star* 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 [6], although this ordering is not embedded within

the shifting process itself. Instead, all possible values are generated first, and then the ordering function is applied to select the POSIX value. This line of thought eventually led us to the 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 with bit sequencing. Inspired by Sulzmann and Lu [8], 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 value, because when a character is matched more than once at the same point, the associated bitsequence may be overwritten. This can cause value erasure and, in some cases, the loss of the POSIX value, as illustrated below in Example 2. We use the bit annotations 0 and 1, similar to the bitcoded derivatives described by Sulzmann and Lu [8].

In contrast to the origianl work by Fischer et al, our function *shift* takes an additional argument, *Bits*. As *shift* is applied, bits are appended to the bitsequence. For example, when shifting through an alternative, 0 is appended to the bitsequence and passed to the left subexpression and 1 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 bitsequence is stored inside this constructor together with the character.

We define the auxiliary function *mkfin* to extract the bitsequence representing the path in bits describing how the expression matched. We adapt *mkeps* from Sulzmann and Lu [8] and from Tan and Urban [9], who define it to construct a value tree and a bitsequence, respectively, for how a nullable expression matches the empty string. Our version also returns a bitsequence: 0 for the left branch and 1 for the right branch in choices, and for *Star* case, 0 indicates the start of an iteration and 1 its end, with the single bit 1 used for the empty-star case. The auxiliary functions *mkfin* and *mkeps* are defined in Figures 14 and 15, while the definition of *fin* is the same as in Section 2.2.2.

We define *shift* for this version as follows:

$$\mathit{shift}(m, c, bs, r) = \begin{cases} \bullet\, r_{\,bs'} & \text{if } c \text{ matches in } r, \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

where $bs'$ is the updated bitsequence. The complete definition appears in Figure 13. The behaviour of each case of the *shift* function is discussed in detail below.

$$\begin{array}{lcl}
\mathit{shift}(m, bs, c, \mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathit{shift}(m, bs, c, \mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{1} \\
\mathit{shift}(m, bs, c, d) & \stackrel{\text{def}}{=} & \begin{cases} \bullet\, d_{bs} & \text{if } m \wedge d = c \\ d & \text{otherwise} \end{cases} \\
\mathit{shift}(m, bs, c, r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{shift}(m,\, bs \oplus 0,\, c,\, r_1) \;+\; \mathit{shift}(m,\, bs \oplus 1,\, c,\, r_2) \\
\mathit{shift}(m, bs, c, r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\begin{cases} \mathit{shift}(m, bs, c, r_1) \cdot \mathit{shift}(\text{true},\, bs \mathbin{@} \mathit{mkeps}(r_1),\, c,\, r_2) & \text{if } m \wedge \mathit{nullable}(r_1) \\ \mathit{shift}(m, bs, c, r_1) \cdot \mathit{shift}(\text{true},\, bs \mathbin{@} \mathit{mkfin}(r_1),\, c,\, r_2) & \text{if } \mathit{fin}(r_1) \\ \mathit{shift}(m, bs, c, r_1) \cdot \mathit{shift}(\text{false}, [\,],\, c,\, r_2) & \text{otherwise} \end{cases}} \\
\mathit{shift}(m, bs, c, r^*) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\begin{cases} (\mathit{shift}(m,\, bs \oplus 0,\, c,\, r))^* & \text{if } m \\ (\mathit{shift}(\text{true},\, bs \mathbin{@} (\mathit{mkfin}(r) \oplus 1),\, c,\, r))^* & \text{if } m \wedge \mathit{fin}(r) \\ (\mathit{shift}(\text{true},\, \mathit{mkfin}(r) \oplus 0,\, c,\, r))^* & \text{if } \mathit{fin}(r) \\ (\mathit{shift}(\text{false}, [\,],\, c,\, r))^* & \text{otherwise} \end{cases}}
\end{array}$$

---

Figure 13: Bit-Annotated *shift* (Version 1). Here, $\oplus$ stands for appending a bit to a bitsequence, while @ stands for concatenation of two bitsequences.

**Character case** $(d)$**:**

- If the input character $c$ matches $d$ and the flag $m$ is true, then a *POINT* wraps the constructor $c$, with the updated bitsequence $bs$ stored inside the *POINT*. Otherwise, the character remains unmarked.

**Alternative case** $(r_1 + r_2)$**:**

- Marks are shifted as before, and the direction of the match is annotated: 0 is added to the bitsequence passed to the left subexpression, and 1 is added to the bitsequence 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 $bs \cup \mathit{mkeps}(r_1)$ is passed to $r_2$, where *mkeps* 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 bitsequence 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, $bs \cup \mathit{mkfin}(r)$ 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, $\mathit{mkfin}(r)$ 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 bitsequence.

$$
\begin{array}{lcl}
\mathit{mkfin}(\bullet_{bs}\, r) & \stackrel{\text{def}}{=} & bs \\
\mathit{mkfin}(\bullet_{bs}\, c) & \stackrel{\text{def}}{=} & bs \\
\mathit{mkfin}(r_1 + r_2) & \stackrel{\text{def}}{=} & \begin{cases} \mathit{mkfin}(r_1) & \text{if } \mathit{fin}(r_1) \\ \mathit{mkfin}(r_2) & \text{otherwise} \end{cases} \\
\mathit{mkfin}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \begin{cases} \mathit{mkfin}(r_1) \mathbin{@} \mathit{mkeps}(r_2) & \text{if } \mathit{fin}(r_1) \wedge \mathit{nullable}(r_2) \\ \mathit{mkfin}(r_2) & \text{otherwise} \end{cases} \\
\mathit{mkfin}(r^*) & \stackrel{\text{def}}{=} & \mathit{mkfin}(r) \mathbin{@} [1]
\end{array}
$$

Figure 14: *mkfin* function

$$
\begin{array}{lcl}
\mathit{mkeps}(\bullet_{bs}\, r) & \stackrel{\text{def}}{=} & bs \\
\mathit{mkeps}(\mathbf{1}) & \stackrel{\text{def}}{=} & [\,] \\
\mathit{mkeps}(r_1 + r_2) & \stackrel{\text{def}}{=} & \begin{cases} 0 \oplus \mathit{mkeps}(r_1) & \text{if } \mathit{nullable}(r_1) \\ 1 \oplus \mathit{mkeps}(r_2) & \text{otherwise} \end{cases} \\
\mathit{mkeps}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{mkeps}(r_1) \mathbin{@} \mathit{mkeps}(r_2) \\
\mathit{mkeps}(r^*) & \stackrel{\text{def}}{=} & [1]
\end{array}
$$

Figure 15: *mkeps* function

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

- String `ba`, regular expression: $(a \cdot b + b \cdot a)$

$$\begin{array}{lcl} \textit{shift}\ b & \rightarrow & (a \cdot b) + (\bullet_{[1]}\, b \cdot a) \\ \\ \textit{shift}\ a & \rightarrow & (a \cdot b) + (b \cdot \bullet_{[1]}\, a) \end{array}$$

Figure 16: Bit-Annotated shift of: $(a \cdot b + b \cdot a)$ with string `ba`

In Example 1 (see Figure 16), the process starts by shifting the first character, b, into the regular expression. Since the expression is an alternative, *shift* is applied to both branches. However, only the right branch matches, because its first subexpression begins with b. A mark is therefore set with the bitcode list [1]. Shifting the second character, a, continues into the concatenation on the right branch. Here the first part is already final, as it matched the preceding b, so the shift moves on to the second part, which matches 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

- String `aaa`, regular expression: $(a + a \cdot a)^*$

$$\begin{array}{lcl} \textit{shift}\ a & \rightarrow & (\bullet_{[0,0]}\, a + \bullet_{[0,1]}\, a \cdot a)^* \\ \\ \textit{shift}\ a & \rightarrow & (\bullet_{[0,0,0,0]}\, a + \bullet_{[0,0,0,1]}\, a \cdot \bullet_{[0,1]}\, a)^* \\ \\ \textit{shift}\ a & \rightarrow & (\bullet_{[0,0,0,0,0,0]}\, a + \bullet_{[0,0,0,0,0,1]}\, a \cdot \bullet_{[0,0,0,1]}\, a)^* \end{array}$$

Figure 17: Bit-Annotated shift of: $(a + a \cdot a)^*$ with string `aaa`

In Example 2 (see Figure 17), after the first shift on a, a mark is placed on the left branch with bits $[0, 0]$, indicating the start of a *Star* 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* iteration followed by a right choice. When the second character is shifted, 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 match, which begins by matching the right-hand side first and then performing another iteration to match the left-hand side. The correct bitsequence in that case would be $[0, 1, 0, 0]$, with the final 1 indicating the end of the *Star* iteration. This behaviour arises because *POINT* stores only a single bitsequence at a time, with no mechanism for preserving multiple marks.

## 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 - Matcher

We initially followed the approach of Fischer et al. [3] and Asperti et al. [2], in which shifting is performed character by character and marks are propagated at each step. However, as in our previous versions, this method made it difficult to maintain an ordering on marks and then to extract the POSIX value.

To address this, we instead let each mark carry the suffix of the input string that is still to be matched, starting with an initial mark carrying the full input string. It gives a more straightforward way of ordering the marks, based on the remaining suffixes. It also appears to offer a closer correspondence with derivatives, in that each mark could be associated with subexpressions created by the derivative, though this point is not yet fully established. The trade-off is that more information must be stored. However, with string-carrying marks, we currently have a better handle on extracting POSIX values.

We define *shifts* in place of *shift* as the function now operates on the full input string, rather than character by character as in the previous versions. The full definition of *shifts* is given in Figure 18. As noted earlier, the empty string $\mathbf{1}$ is not marked. This choice ensures termination of the *shifts* function,

particularly in the *Star* case: if $\mathbf{1}$ were allowed to produce marks, it would lead to infinite unfolding and non-termination.

The shifting process starts with a list of marks containing one initial mark carrying the full input string. As the mark is shifted through the expression, new marks may be produced depending on its structure, such as in sequences, stars, or alternatives. For example, in the case of an alternative, the mark is passed to each branch, and the resulting lists of marks are combined. Each time a character matches, it is stripped from the corresponding string carried by the mark. If a character does not match the prefix of the string carried by a mark, that mark will be deleted. A match occurs at the end of shifting when an empty mark has been produced, indicating that all characters have been matched. When the input string itself is empty, *shifts* is not applied; instead, the *nullable* function (with the same definition as in Section 2.2.2, shown in Figure 12) is used to check whether the regular expression accepts the empty string.

The number of marks when using lists is, in most cases, linear in the input length. The exception is the *Star* case, where the total number of marks can grow exponentially or more. This proliferation arises when the inner $r$ contains alternatives and sequences that may add or combine lists of marks, which are then fed back into the *Star* through repeated unfoldings. Even though the length of individual marks becomes shorter with each successful shift, their number still grows in this way. In such cases, if the regular expression has $m$ branch choices, then on an input of length $n$ the number of marks can reach $\mathcal{O}(m^n)$. For example, for the regular expression $(a + a)^*$ and input of the form $\underbrace{a \cdots a}_{n}$, we have $m = 2$. For $n = 3$, the first three unfoldings are as follows:

$$\begin{aligned}
&\text{step 1}: \quad \bullet\, aa \;@\; \bullet\, aa \quad (2)\\
&\text{step 2}: \quad \bullet\, a \;@\; \bullet\, a \;@\; \bullet\, a \;@\; \bullet\, a \quad (4)\\
&\text{step 3}: \quad \underbrace{\bullet\,[\,] \;@\; \cdots \;@\; \bullet\,[\,]}_{8 \text{ marks}}
\end{aligned}$$

Using sets, on the other hand, removes duplicates, which are the reason lists can grow exponentially (or more) . The number of distinct marks therefore remains linear in the input string length. In the same example we obtain only $\{\bullet\, aa\} \to \{\bullet\, a\} \to \{\bullet\,[\,]\}$.

We started with a lexer implementation of this version, and from initial

testing it produces the POSIX value, yet this remains to be formally proven. In that version we introduce reshuffling, where marks are ordered based on their remaining strings. At present, the reordering is carried out in the *Sequence* $(r_1 \cdot r_2)$ case after shifting through the first part.

The behaviour of each case of the *shifts* function is discussed in detail below.

$$
\begin{array}{rcl}
\mathit{shifts}(ms, 0) & \stackrel{\text{def}}{=} & [\,] \\
\mathit{shifts}(ms, 1) & \stackrel{\text{def}}{=} & [\,] \\
\mathit{shifts}(ms, d) & \stackrel{\text{def}}{=} & [\,\bullet\, s \mid \bullet\, d :: s \in ms\,] \\
\mathit{shifts}(ms, r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{shifts}(ms, r_1) \mathbin{@} \mathit{shifts}(ms, r_2) \\
\mathit{shifts}(ms, r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \text{let } ms' = \mathit{shifts}(ms, r_1) \text{ in} \\
& & \begin{cases}
\mathit{shifts}(ms' \mathbin{@} ms, r_2) \mathbin{@} ms' & \text{if } \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2), \\
\mathit{shifts}(ms' \mathbin{@} ms, r_2) & \text{if } \mathit{nullable}(r_1), \\
\mathit{shifts}(ms', r_2) \mathbin{@} ms' & \text{if } \mathit{nullable}(r_2), \\
\mathit{shifts}(ms', r_2) & \text{otherwise}
\end{cases} \\
\mathit{shifts}(ms, r^*) & \stackrel{\text{def}}{=} & \text{let } ms' = \mathit{shifts}(ms, r) \text{ in} \\
& & \text{if } ms' = [\,] \text{ then } [\,] \text{ else } \mathit{shifts}(ms', r^*) \mathbin{@} ms'
\end{array}
$$

Figure 18: *shifts* function. A mark is represented as $\bullet\, s$, where $s$ is a string, and $ms$ is a list of such marks.

**Character case** $(c)$**:**

- If the head of the string in a mark matches $c$, it is stripped and the remainder is kept; otherwise the mark is dropped.

**Alternative case** $(r_1 + r_2)$**:**

- The list of marks $ms$ is shifted into both subexpressions $r_1$ and $r_2$. The two resulting lists are then concatenated, which corresponds to matching with either $r_1$ or $r_2$.

**Sequence case** $(r_1 \cdot r_2)$**:**

- This is the most elaborate case. It begins by shifting the marks into the first part of the sequence, $r_1$, producing a new list of marks $ms'$. The result then depends on several conditions:, namely:

  1. $\mathit{nullable}(r_1 \wedge r_2)$: both $r_1$ and $r_2$ are nullable, so both can be skipped and the shifting must account for this by combining the results appropriately. To account for $r_1$ being skipped, the original marks $ms$ are appended to $ms'$ and then shifted into $r_2$. In this way, skipping $r_1$ has the same effect as shifting $ms$ directly into $r_2$. Finally, $ms'$ itself is appended to the result to account for skipping $r_2$, which corresponds to matching with $r_1$ only.
  2. $\mathit{nullable}(r_1)$: if $r_1$ is nullable, it may be skipped. In this case, the original marks $ms$ are appended to $ms'$ and the result is then shifted into $r_2$, which corresponds to matching directly with $r_2$.
  3. $\mathit{nullable}(r_2)$: if $r_2$ is nullable, it may be skipped. Here, $ms'$ is shifted into $r_2$, and the result is then appended with $ms'$ to account for the case where $r_2$ is skipped, which corresponds to matching with $r_1$ only.
  4. $\neg\mathit{nullable}(r_1) \wedge \neg\mathit{nullable}(r_2)$: if neither $r_1$ nor $r_2$ is nullable, the marks $ms'$ are shifted directly into $r_2$.

**Star case** $(r^*)$**:**

- The list of marks $ms$ is first shifted into $r$, producing a new list $ms'$. If $ms'$ is empty, this corresponds to the end of the iterations of the *Star*, and the result is the empty list. Otherwise, $ms'$ records one completed iteration and is appended to the result of shifting $ms'$ into the *Star*, corresponding to adding a new iteration.

The matcher is defined as follows.

$$\mathit{matcher}(s, r) \stackrel{\text{def}}{=} \begin{cases} \mathit{nullable}(r) & \text{if } s = [\,], \\ \bullet\,[] \in \mathit{shifts}([\bullet\, s], r) & \text{otherwise.} \end{cases}$$

To illustrate the behaviour of this version, we consider three examples. The first uses the regular expression $(a + (a \cdot c))$ and string `ac`. We show step by step how the marks are reduced as characters are stripped and non-matching paths are dropped. The second example considers the *Star* case, highlighting how the marks are unfolded. A third example, $(a^* \cdot a^*)$ with string `aa`, shows that, in this version, all possible marks must be generated, since one of them may be the only one required to complete the match when the *Star* expression is part of a larger expression.

In Example 1 (Figure 19), we consider the regular expression $(a + (a \cdot c))$ and the string `ac`. Matching begins with the initial mark carrying the full string, which is then propagated into the two subexpressions: the left subexpression $a$ and the right subexpression $(a \cdot c)$. In the left subexpression $a$, the initial character matches, so it is stripped from the mark, leaving the string `c`. In the right subexpression $(a \cdot c)$, the sequence matches both characters, so the string in the mark is reduced to the empty string. At this point, the list of marks $[\bullet_{[c]}, \bullet_{[]}]$ is obtained, and with no more *shifts* calls, and one mark reduced to the empty string, the expression is accepted.

In Example 2 (Figure 20), we consider the regular expression $(a^*)$ and the string `aaa`. For the *Star* case, the process starts by shifting the received marks into the inner expression; here, the initial mark is shifted into $a$. When the character $a$ matches, the leading $a$ is stripped from the string carried by the mark, and the inner *shifts* call produces a new mark with this shorter string. If the result is empty—meaning that no character was consumed—no further iteration takes place. This mechanism has two effects: first, the strings carried by the marks either remain unchanged or become shorter;in the sense that their length decreases. Second, it enumerates all the ways the *Star* expression can match the input, from no consumption to consuming the entire string. The algorithm therefore returns a list of marks representing these alternatives, and, as Example 3 will show, one of these marks may be the only one required to complete the match when the *Star* expression is part of a larger expression. This observation also suggests a possible optimisation: rather than generating all marks, it may be sufficient to generate only those needed in a given case.

In Example 3 (Figure 21), we consider the regular expression $((a^* \cdot a) \cdot a)$ and the string `aaa`. The full string will be in the initial mark as before. Then the mark will be passed to the first part of the outer sequence, which is itself a sequence. In step 2, the mark will then enter the inner sequence's first part, $r_1$, which is $a^*$, and will produce the list of marks $[\bullet\, aa, \bullet\, a, \bullet_{[]}]$, as shown

in Example 2. However, since $r_1$ is a *Star* and thus nullable, the original mark $[\bullet\, aaa]$ is appended to the result list of the first part, to account for the possibility of skipping $r_1$. Then this list of marks will be passed on, in step 3, to the second part of the inner sequence, resulting in the consumption of a character from each of the marks (and also dropping the empty-list mark from $r_1$, since it cannot match the character in $r_2$). In step 4, the resulting list will be passed to the second part of the outer sequence, consuming another character from the remaining marks (and also dropping the empty-list mark), resulting in the final list of marks $[\bullet_{\,[\,]}, \bullet\, a]$. In this example, we see that the *Star* part of the expression produces multiple marks, and one of them is necessary to complete the match for the entire expression, in particular the mark $\bullet\, aa$, which represents consuming only one character by the *Star* constructor.

- String `ac`, regular expression: $(a + (a \cdot c))$
  1. $\big|_{[\bullet\, ac]}\ (a + (a \cdot c))$
  2. $(\big|_{[\bullet\, ac]} a + \big|_{[\bullet\, ac]} (a \cdot c))$
  3. $(a\ \big|_{[\bullet\, c]} + (a \cdot c)\ \big|_{[\bullet_{\,[\,]}]})$

---

Figure 19: String-carrying shifts for: $(a + (a \cdot c))$ with string `ac`

- String `aaa`, regular expression: $(a^*)$
  1. $\big|_{[\bullet\, aaa]}\ (a^*)$
  2. $(\big|_{[\bullet\, aa]} a^*)$
  3. $(\big|_{[\bullet\, aa, \bullet\, a]} a^*)$
  4. $(a^*\ \big|_{[\bullet\, aa, \bullet\, a, \bullet_{\,[\,]}]})$

---

Figure 20: String-carrying shifts for: $(a^*)$ with string `aaa`

- String `aaa`, regular expression: $((a^* \cdot a) \cdot a)$

  1. $\big|_{[\bullet aaa]}\ ((a^* \cdot a) \cdot a)$
  2. $((a^* \big|_{[\bullet aa, \bullet a, \bullet [\,], \bullet aaa]} \cdot a) \cdot a)$
  3. $((a^* \cdot a \big|_{[\bullet a, \bullet [\,], \bullet aa]}) \cdot a)$
  4. $((a^* \cdot a) \cdot a) \big|_{[\bullet [\,], \bullet a]}$

---

Figure 21: String-carrying shifts for: $((a^* \cdot a) \cdot a)$ with string `aaa`

### 3.4 String-Carrying Marks - Lexer **so far**

To be Added Later.

# 4 Future Work

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 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 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190, 2009.

[8] M. Sulzmann and K. F. Lu. Posix regular expression parsing with derivatives. *Science of Computer Programming*, 89:179–193, 2014.

[9] C. Tan and C. Urban. *POSIX Lexing with Bitcoded Derivatives*, pages 26:1–26:18. Apr. 2023.