# Progress Report - 9 Months

Meshal Binnasban

October 29, 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 "size explosion". Sulzmann and Lu [9] extended derivatives with bitcodes to record lexing information, but the "size explosion" 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 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

The notion of derivatives in regular expressions was introduced by Brzozowski in 1964 [3], but only in the last decade has it been rediscovered and gained renewed attention—for example in the work of Owens [8] and Might [5]. This renewed interest is largely due to the method's conceptual simplicity and the ease with which it can be implemented in functional programming languages. However, derivatives suffer from "size explosion", 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. 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 [9], followed by background on the regular expression matchers based on marks described by Fischer et al. [4] 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

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

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 $a$.

$$
\begin{aligned}
der_a(\mathbf{0}) &\stackrel{\text{def}}{=} \mathbf{0} \\[4pt]
der_a(\mathbf{1}) &\stackrel{\text{def}}{=} \mathbf{0} \\[4pt]
der_a(c) &\stackrel{\text{def}}{=} \begin{cases} \mathbf{1} & \text{if } a = c \\ \mathbf{0} & \text{if } a \neq c \end{cases} \\[4pt]
der_a(r_1 + r_2) &\stackrel{\text{def}}{=} der_a(r_1) + der_a(r_2) \\[4pt]
der_a(r_1 \cdot r_2) &\stackrel{\text{def}}{=} der_a(r_1) \cdot r_2 \; + \; \begin{cases} der_a(r_2) & \text{if } \varepsilon \in L(r_1) \\ \mathbf{0} & \text{otherwise} \end{cases} \\[4pt]
der_a(r^*) &\stackrel{\text{def}}{=} der_a(r) \cdot r^*
\end{aligned}
$$

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 could simplify it to $\mathbf{0}$ which can match the same strings, namely none (see Figure 3). 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 the overall size of the regular expression. Such simplifications are sometimes necessary in the derivative method in order to keep the size manageable but even though best effort still means that the size can explode.

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

$$\begin{aligned}
der_a\,(der_b\, r) \;&=\; der_a\,(\mathbf{0} \cdot b + \mathbf{1} \cdot a) \\
&=\; der_a\,(\mathbf{0} \cdot b) + der_a\,(\mathbf{1} \cdot a) \\
&=\; der_a\,(\mathbf{0}) \cdot b + (der_a\,(\mathbf{1}) \cdot a + der_a\, a) \\
&=\; \mathbf{0} \cdot b + (\mathbf{0} \cdot a + \mathbf{1})
\end{aligned}$$

---

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 and increase the size 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 [9] 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, for example, the regular expression $(a + aa)^*$ and the derivative with respect to a single $a$.

$$\begin{aligned}
der_a\,(a + aa)^* \;&=\; (der_a\,(a + aa)) \cdot (a + aa)^* \\
&=\; (der_a\, a + der_a\,(aa)) \cdot (a + aa)^* \\
&=\; (\mathbf{1} + der_a\,(a \cdot a)) \cdot (a + aa)^* \\
&=\; (\mathbf{1} + (\mathbf{1} \cdot a)) \cdot (a + aa)^* \\
&=\; (\mathbf{1} + a) \cdot (a + aa)^*
\end{aligned}$$

Each derivative step may introduce new structure, unfolding all possible ways in which the $r^*$ expression can match the input.

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 [9, 10], but it does not completely solve the underlying problem. Even with aggressive simplifications—as in Sulzmann and Lu's bitcoded approach 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 derivatives 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" 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 above to demonstrate the impact of the "size explosion" 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.

### 2.1.2 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, 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 [9]. 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 during matching. Figure 6 illustrates how bitcoding works with $(a+ab)(b+\mathbf{1})$ on the string `ab`, proceeding step by step by constructing bitcoded derivatives and tracking how the bitcode grows. Initially, the algorithm internalizes the regular expression by applying the function *intern* (Figure 7), adding bit annotations to the alternative constructors in the expression: 0 for left branches and 1 for right branches [9]. This annotation process is implemented by the function *fuse* (Figure 8). The bitcode is updated while taking the derivatives to reflect the choices made during matching. The function *mkeps*(Figure 9), as defined by Sulzmann and Lu [9], extracts the bitcode from nullable derivatives. A regular expression is nullable if it can match the empty string.

Derivatives can cause the regular expression to expand. In particular, the sequence case expands into an alternative when the first part is nullable (see Figure 6, Step 3, where the concatenation becomes nullable). This accounts for the situation in which the first part, $r_1$, may be skipped during matching: the left branch assumes $r_1$ is not skipped, while the right branch assumes it is skipped and instead takes the derivative of $r_2$ directly. This behaviour also illustrates why derivatives tend to grow in size. Sulzmann and Lu [9] 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 Figure 6, there are two possible matches, but the algorithm prefers the left one, $[1, 1]$, 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 $r_2$, which matches the empty string. Decoding this against the original expression yields the POSIX value

$$Seq\,(\,Right\,(\,Seq\,(\,a,b\,)),Right\,(\,Empty\,))\,.$$

## 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 [6], 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. [4] 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. [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 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 atoms matching the current input symbol. The main transition function is therefore written as

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

By contrast, Nipkow and Traytel show that the work of Asperti et al., although presented as McNaughton–Yamada, is in fact a dual construction: mark-before-atom. While Asperti et al. described their approach as following the McNaughton–Yamada–Glushkov framework, their work actually corresponds to the opposite variant, where marks are placed before reading the next 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 therefore the inverse of the previous one:

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

### 2.2.1 Motivation for a Marked Approach

The marked approach offers a promising alternative to derivatives for regular expression matching by addressing some of the limitations found in the derivative method. The derivative method provides a more direct way of matching compared to constructing an NFA and then a DFA, which can suffer from exponential state explosion, up to $2^n$. Engines that construct DFAs, such as those used in Rust, attempt to mitigate this problem by building the automaton dynamically: they start from the initial state and construct additional DFA states only as required by the input string. This avoids generating the full automaton up front, but even so, it can still lead to large memory usage for expressions such as $r^n$, which can effectively require $n$ copies of the automaton in the worst case. Other engines, such as those in Python, use NFAs instead of DFAs. This design allows for additional features, such as backreferences and lookarounds, but may result in catastrophic backtracking when the engine needs to explore a large portion of the automaton—or even the entire automaton—before finding a match or rejecting an input. The derivative approach, albeit elegant in design and supported by formal proofs—unlike many existing engines that lack such guarantees—can still grow quickly in size, as each derivative may add new subexpressions at each step. Even with aggressive simplifications, these subexpressions cannot always be simplified when they occur at different levels of the expression tree.

The marked approach relies on propagating marks within the regular expression rather than constructing new subexpressions. Our main motivation is that this method could support efficient 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. 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 extended constructors, 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 dual construction to the post-mark algorithm of Fischer et al.; moreover, the former is a homomorphic image of the latter and thus never larger in number of states. Consequently, we use Fischer et al.'s approach as the foundation for extending the marked approach to matching with POSIX value extraction.

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

$$\begin{aligned}
der_a\,(der_a\, r) &= der_a\,((a)^* + (a \cdot (aa)^*) + (aa \cdot (aaa)^*) + \ldots) \cdot r^* + der_a\, r^* \\
&= (\mathbf{1} \cdot (a)^* + (\mathbf{1} \cdot (aa)^*) + ((\mathbf{1} \cdot a) \cdot (aaa)^*) + \ldots) \cdot r^* + \\
&\quad\ ((a)^* + (a \cdot (aa)^*) + (aa \cdot (aaa)^*) + \ldots) \cdot r^* \\
&= ((a)^* + (aa)^* + (a \cdot (aaa)^*) + \ldots) \cdot r^* + \\
&\quad\ ((a)^* + (a \cdot (aa)^*) + (aa \cdot (aaa)^*) + \ldots) \cdot r^*
\end{aligned}$$

$$\begin{aligned}
der_a\,(der_a\,(der_a\, r)) &= der_a\,(((a)^* + (aa)^* + (a \cdot (aaa)^*) + \ldots) \cdot r^*) + \\
&\quad\ der_a\,(((a)^* + (a \cdot (aa)^*) + (aa \cdot (aaa)^*) + \ldots) \cdot r^*) \\
&= (der_a\,(((a)^* + ((aa)^*) + (a \cdot (aaa)^*) + \ldots) \cdot r^*) + der_a\, r^*) + \\
&\quad\ (der_a\,(((a)^* + (a \cdot (aa)^*) + (aa \cdot (aaa)^*) + \ldots) \cdot r^*) + der_a\, r^*)
\end{aligned}$$

Figure 4: Derivatives of $((a)^* + (aa)^* + (aaa)^* + (aaaa)^* + (aaaaa)^*)^*$ with respect to the string `aaa`. Each step introduces additional copies of the original regular expression at different derivative levels, where simplifications cannot be applied. For example, after the second step, $(a)^*$ appears in both alternatives, but—being under different products with $r$—the two occurrences cannot be merged.
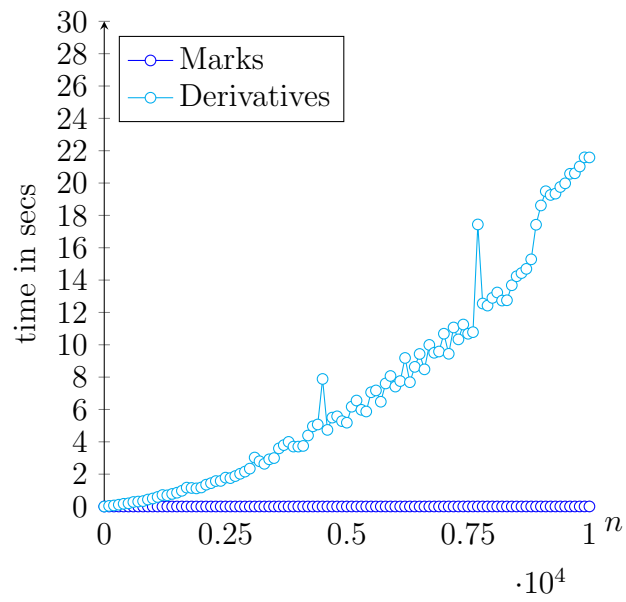
Figure 5: Runtime comparison of derivative-based and mark-based regular expression matchers for $((a)^* + (aa)^* + (aaa)^* + (aaaa)^* + \dots)^*$ with input strings of length $n$.

1. Step 1: Internalizing the regular expression

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

2. Step 2: input a

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

3. Step 3: input b

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

---

Figure 6: Step-by-step construction of bitcoded derivatives for $(a + ab) \cdot (b + \mathbf{1})$ with input $\mathtt{ab}$. Step 1 shows the result of internalization using *fuse*, where alternatives are annotated with bits. Step 2 takes the derivative with respect to $a$; the two branches of the result are nullable at this stage: the left branch of $r_1$ and the right branch of $r_2$ can both match the empty string after consuming the first character, corresponding to the bitcodes $[0]$ and $[1]$ respectively. Step 3 takes the derivative with respect to $b$, expanding the expression due to the nullable concatenation. Finally, *mkeps* extracts the bitcode $[1, 1]$.

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

Figure 7: The *intern* function, which internalizes regular expressions by adding bit annotations to alternatives, as defined by Sulzmann and Lu [9].

$$
\begin{aligned}
fuse\ bs\ (c_{bs'}) &\overset{\text{def}}{=} c_{(bs\ @\ bs')} \\
fuse\ bs\ ((r_1 + r_2)_{bs'}) &\overset{\text{def}}{=} (r_1 + r_2)_{(bs\ @\ bs')} \\
fuse\ bs\ ((r_1 \cdot r_2)_{bs'}) &\overset{\text{def}}{=} (r_1 \cdot r_2)_{(bs\ @\ bs')} \\
fuse\ bs\ (r^*{}_{bs'}) &\overset{\text{def}}{=} (r^*)_{(bs\ @\ bs')} \\
fuse\ bs\ (r^n{}_{bs'}) &\overset{\text{def}}{=} (r^n)_{(bs\ @\ bs')}
\end{aligned}
$$

Figure 8: The *fuse* function, which adds bit annotations to expressions, as defined by Sulzmann and Lu [9].

$$mkeps(\mathbf{1}_{bs}) \quad \overset{\text{def}}{=} \quad bs$$

$$mkeps((r_1 + r_2)_{bs}) \quad \overset{\text{def}}{=} \quad \begin{cases} bs \ @ \ mkeps(r_1) & \text{if } nullable(r_1) \\ bs \ @ \ mkeps(r_2) & \text{otherwise} \end{cases}$$

$$mkeps((r_1 \cdot r_2)_{bs}) \quad \overset{\text{def}}{=} \quad bs \ @ \ mkeps(r_1) \ @ \ mkeps(r_2)$$

$$mkeps((r^*)_{bs}) \quad \overset{\text{def}}{=} \quad bs \ @ \ [1]$$

Figure 9: The *mkeps* function, which extracts bit annotations from nullable expressions, as defined by Sulzmann and Lu [9].

# 3  Our Approach

We began our work by implementing the marked approach described by Fischer et al. [4] in Scala; details are provided in Appendix A.1. 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 value. This occurs when marks are overwritten after a character is matched more than once; further details are provided in Appendix A.2.

The second version was developed to address two main issues: 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 bit sequence that could lead to acceptance. Through reasoning and testing, we refined 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 [7], 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; further details are provided in Appendix A.3.

This line of thought eventually led us to the third version of the algorithm, which uses string-annotated marks. In this version, each mark carries the suffix of the input string that is still to be matched. This allows us to compare marks based on their remaining suffixes. We also began annotating these marks with bit sequences to record the choices made during matching. Although this version is not yet finished, the initial results are promising. The remainder of this section describes the third (latest) version; the earlier versions are presented in the Appendix.

## 3.1 String-Carrying Marks - Matcher

We initially followed the approach of Fischer et al. [4] 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 10. As discussed in Appendix A.1, the empty string expression **1** is not marked. This choice ensures termination of the *shifts* function, particularly in the *Star* case: if **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 A.1, shown in Figure 16) 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

19

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, where @ stands for list concatenation.

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

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 $\{\bullet\, aa \ \rightarrow\, \bullet\, a \ \rightarrow\, \bullet\,[]\}$. When marks are represented only by their remaining suffixes, there can be at most $n$ distinct marks for an input string of length $n$, since each mark corresponds to a unique suffix of the input and there are at most $n$ such suffixes. Even in the presence of nested stars, the outer unfolding reuses the same set of suffix marks generated by the inner star, so although the number of recursive calls can increase, the number of distinct marks at each stage remains bounded by $n$.

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 case after shifting through the first part and in the *Star* and $r^n$ after shifting to the inner regex inside.

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

**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.

$$shifts(ms, 0) \quad \overset{\text{def}}{=} \quad [\,]$$

$$shifts(ms, 1) \quad \overset{\text{def}}{=} \quad [\,]$$

$$shifts(ms, d) \quad \overset{\text{def}}{=} \quad [\, \bullet s \mid \bullet d :: s \in ms \,]$$

$$shifts(ms, r_1 + r_2) \quad \overset{\text{def}}{=} \quad shifts(ms, r_1) \,@\, shifts(ms, r_2)$$

$$shifts(ms, r_1 \cdot r_2) \quad \overset{\text{def}}{=} \quad \text{let } ms' = shifts(ms, r_1) \text{ in}$$

$$\begin{cases} shifts(ms' \,@\, ms, r_2) \,@\, ms' & \text{if } nullable(r_1) \wedge nullable(r_2), \\ shifts(ms' \,@\, ms, r_2) & \text{if } nullable(r_1), \\ shifts(ms', r_2) \,@\, ms' & \text{if } nullable(r_2), \\ shifts(ms', r_2) & \text{otherwise} \end{cases}$$

$$shifts(ms, r^*) \quad \overset{\text{def}}{=} \quad \text{let } ms' = shifts(ms, r) \text{ in}$$

$$\text{if } ms' = [\,] \text{ then } [\,] \text{ else } shifts(ms', r^*) \,@\, ms'$$

Figure 10: *shifts* function. Marks are annotated with strings. A mark is represented as $\bullet s$, where $s$ is a string, and $ms$ is a list of such marks.

**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. $nullable(r_1) \wedge nullable(r_2)$: 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 $r_2$.

21

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. *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. *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. $\neg$*nullable*$(r_1) \wedge \neg$*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.

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

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 11), 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

22

$a$, 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 further calls to *shifts* and one mark reduced to the empty string, the expression is accepted.

In Example 2 (Figure 12), 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 13), 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 11: Example 1: String-carrying shifts for: $(a + (a \cdot c))$ with the 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 12: Example 2: String-carrying shifts for: $(a^*)$ with the 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 13: Example 3: String-carrying shifts for: $((a^* \cdot a) \cdot a)$ with string `aaa`

# 4  Future Work

The development of the marked matcher is still ongoing, and several directions for future work have been identified:

- **Continue the latest version to produce POSIX values.** Work has started on POSIX value extraction for the latest version of the matcher. Initial results indicate that it is producing POSIX values. Further work is needed to fix the handling of exact repetition $r^n$.

- **Extending support for additional operators.** Future work will focus on extending the matcher to handle intersection and negation operators. This will require extending both the shifting function and the bitcoding scheme so that these operators are handled correctly during matching.

- **Formal verification in Isabelle/HOL.** Work has started on formally proving the correctness of the latest version of the algorithm in Isabelle/HOL. The next step is to prove that it always produces the POSIX value.

- **Performance and optimisation.** We are exploring ways to make the algorithm more efficient. We plan to investigate strategies to control or reduce mark proliferation in complex expressions and to handle bitcode annotations more efficiently.

# 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 reexamined. *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.

# A  Appendix

## A.1  Scala Implementation of Fischer et al.'s Marked Approach

In Fischer et al.'s 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.

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

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* 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.

---

[1] Our choice follows Fischer et al. [4], where $\mathbf{1}$ 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.

$$shift(m, c, \mathbf{0}) \quad \overset{\text{def}}{=} \quad \mathbf{0}$$

$$shift(m, c, \mathbf{1}) \quad \overset{\text{def}}{=} \quad \mathbf{1}$$

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

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

$$shift(m, c, r_1 \cdot r_2) \quad \overset{\text{def}}{=}$$

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

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

Figure 14: The *shift* function in the Scala implementation of Fischer's Marked Approach.

- **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 $fin(r_1)$ 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 $fin(r)$ holds.

The formal definitions of the auxiliary functions $fin$ and $nullable$ appear in Figures 15 and 16, as defined by Fischer et al. [4].

$$fin(\mathbf{0}) \quad \overset{\text{def}}{=} \quad \text{false}$$

$$fin(\mathbf{1}) \quad \overset{\text{def}}{=} \quad \text{false}$$

$$fin(c) \quad \overset{\text{def}}{=} \quad \text{false}$$

$$fin(\bullet\, c) \quad \overset{\text{def}}{=} \quad \text{true}$$

$$fin(r_1 + r_2) \quad \overset{\text{def}}{=} \quad fin(r_1) \vee fin(r_2)$$

$$fin(r_1 \cdot r_2) \quad \overset{\text{def}}{=} \quad (fin(r_1) \wedge nullable(r_2)) \vee fin(r_2)$$

$$fin(r^*) \quad \overset{\text{def}}{=} \quad fin(r)$$

Figure 15: The $fin$ function, which checks whether a marked expression is in a final state

## A.2 Bit-Annotated Version 1

In this version, we extended the marked approach with bit sequencing. 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 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 [9].

In contrast to the original 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

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

Figure 16: The *nullable* function, which checks whether an expression can match the empty string.

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 [9] and from Tan and Urban [10], 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 18 and 19, while the definition of *fin* is the same as in Section A.1. We define *shift* for this version as follows:

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

where $bs'$ is the updated bitsequence. The complete definition appears in Figure 17. The behaviour of each case of the *shift* function is discussed in detail below.

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

$$
\begin{aligned}
shift(m, bs, c, \mathbf{0}) \quad &\overset{\text{def}}{=} \quad \mathbf{0} \\[4pt]
shift(m, bs, c, \mathbf{1}) \quad &\overset{\text{def}}{=} \quad \mathbf{1} \\[4pt]
shift(m, bs, c, d) \quad &\overset{\text{def}}{=} \quad
\begin{cases}
\bullet \, d_{bs} & \text{if } m \wedge d = c \\
d & \text{otherwise}
\end{cases} \\[10pt]
shift(m, bs, c, r_1 + r_2) \quad &\overset{\text{def}}{=} \quad shift(m,\, bs \oplus 0,\, c,\, r_1) \; + \; shift(m,\, bs \oplus 1,\, c,\, r_2) \\[4pt]
shift(m, bs, c, r_1 \cdot r_2) \quad &\overset{\text{def}}{=}
\end{aligned}
$$

$$
\begin{cases}
shift(m, bs, c, r_1) \cdot shift(\text{true}, bs \,@\, mkeps(r_1),\, c,\, r_2) & \text{if } m \wedge nullable(r_1) \\
shift(m, bs, c, r_1) \cdot shift(\text{true}, bs \,@\, mkfin(r_1),\, c,\, r_2) & \text{if } fin(r_1) \\
shift(m, bs, c, r_1) \cdot shift(\text{false}, [\,],\, c,\, r_2) & \text{otherwise}
\end{cases}
$$

$$
shift(m, bs, c, r^*) \quad \overset{\text{def}}{=}
$$

$$
\begin{cases}
(shift(m,\, bs \oplus 0,\, c,\, r))^* & \text{if } m \\
(shift(\text{true},\, bs \,@\, (mkfin(r) \oplus 1),\, c,\, r))^* & \text{if } m \wedge fin(r) \\
(shift(\text{true},\, mkfin(r) \oplus 0,\, c,\, r))^* & \text{if } fin(r) \\
(shift(\text{false}, [\,],\, c,\, r))^* & \text{otherwise}
\end{cases}
$$

---

Figure 17: Bit-Annotated Version 1 *shift*. Here, $\oplus$ stands for appending a bit to a bitsequence, while $@$ stands for concatenation of two bitsequences.

- 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 $(r_1 \cdot r_2)$:**

31

- 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 $bs \cup mkeps(r_1)$ 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, $bs \cup mkfin(r)$ 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 bitsequence.

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.

In Example 1 (see Figure 20), 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

$$mkfin(\bullet_{bs} r) \quad \stackrel{\text{def}}{=} \quad bs$$

$$mkfin(\bullet_{bs} c) \quad \stackrel{\text{def}}{=} \quad bs$$

$$mkfin(r_1 + r_2) \quad \stackrel{\text{def}}{=} \quad \begin{cases} mkfin(r_1) & \text{if } fin(r_1) \\ mkfin(r_2) & \text{otherwise} \end{cases}$$

$$mkfin(r_1 \cdot r_2) \quad \stackrel{\text{def}}{=} \quad \begin{cases} mkfin(r_1) \,@\, mkeps(r_2) & \text{if } fin(r_1) \wedge nullable(r_2) \\ mkfin(r_2) & \text{otherwise} \end{cases}$$

$$mkfin(r^*) \quad \stackrel{\text{def}}{=} \quad mkfin(r) \oplus 1$$

Figure 18: *mkfin* function

$$mkeps(\mathbf{1}) \quad \stackrel{\text{def}}{=} \quad []$$

$$mkeps(r_1 + r_2) \quad \stackrel{\text{def}}{=} \quad \begin{cases} 0 \oplus mkeps(r_1) & \text{if } nullable(r_1) \\ 1 \oplus mkeps(r_2) & \text{otherwise} \end{cases}$$

$$mkeps(r_1 \cdot r_2) \quad \stackrel{\text{def}}{=} \quad mkeps(r_1) \,@\, mkeps(r_2)$$

$$mkeps(r^*) \quad \stackrel{\text{def}}{=} \quad [1]$$

Figure 19: *mkeps* function

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 expression. The resulting bitsequence is [1], recording that the match followed the right branch of the alternative.

In Example 2 (see Figure 21), 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

33

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

$$shift\ b \quad \rightarrow \quad (a \cdot b) + (\bullet_{[1]} b \cdot a)$$

$$shift\ a \quad \rightarrow \quad (a \cdot b) + (b \cdot \bullet_{[1]} a)$$

---

Figure 20: Example 2: Bit-Annotated Version 1 shift of: $(a \cdot b + b \cdot a)$ with the string `ba`

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

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

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

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

---

Figure 21: Example 2: Bit-Annotated Version 1 shift of: $(a + a \cdot a)^*$ with the string `aaa`

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.

## A.3  Bit-Annotated Version 2

In this version, we updated the previous Bit-Annotated Version 1 to accumulate all possible bitsequences leading to a match, rather than storing only a single bitsequence at each marked position. This change was motivated by two issues observed in Version 1: the overwriting of marks when a character is matched more than once at the same position, which can cause the POSIX value to be lost, and the absence of a mechanism to order marks during

shifting so as to always produce the POSIX value.

To address the first issue, each marked position now carries a list of bitsequences. Whenever a character is matched, instead of overwriting the existing bitsequence, the new bitsequence is appended to the list already stored at that position.

To address the second issue, ordering is applied after the shifting process, using a function inspired by the work of Okui and Suzuki [7]. This ordering is not embedded within shifting itself: all possible values are generated first and then ordered externally.

The definition of *shift* for this version is shown in Figure 22. The key difference in this version, compared to Version 1, is that all possible paths are considered. For example, in the sequence case, the bitsequences that arise when $r_1$ is nullable and can be skipped are passed along together with the bitsequences that arise when $r_1$ is in a final position. This means that the right-hand side of a sequence ($r_2$) receives both lists of bitsequences — those corresponding to $r_1$ being skipped and those where $r_1$ is final — ensuring that all possible paths are accounted for. Similarly, the function *mkfin* is updated to return all possible bitsequences.

The auxiliary function *mkfin* is updated to return lists of bits, while *mkeps* remains unchanged from the previous version. The updated definition of *mkfin* is given in Figure 23.

By accumulating lists of bitsequences at each position, no potential path is lost due to overwriting. Although a formal proof is not yet complete, substantial testing and reasoning give us confidence that this version generates all possible values for matching a string against a regular expression, including the POSIX value. However, this comes at the cost of an increase in the number of values returned, especially in the *Star* case, where every possible way of matching is generated.

## A.4   String-Carrying Marks - Lexer (so far)

In this version, we extend the string-carrying-marks design by attaching a bit sequence to each mark, so that each mark consists of its remaining suffix and the corresponding bit sequence. Similar to the Bit-Annotated Version 1 and 2, the bit sequences are built incrementally as the marks are shifted through the regular expression. The exception now is that the order of marks is based on their remaining suffixes: shorter suffixes are prioritised over longer ones, and marks with the same suffix length preserve their original order.

$$shift(m, bs, c, \mathbf{0}) \quad \overset{\text{def}}{=} \quad \mathbf{0}$$

$$shift(m, bs, c, \mathbf{1}) \quad \overset{\text{def}}{=} \quad \mathbf{1}$$

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

$$shift(m, bs, c, r_1 + r_2) \quad \overset{\text{def}}{=} \quad shift(m,\, bs \oplus 0,\, c,\, r_1) \;+\; shift(m,\, bs \oplus 1,\, c,\, r_2)$$

$$shift(m, bs, c, r_1 \cdot r_2) \quad \overset{\text{def}}{=} \quad shift(m, bs, c, r_1) \cdot$$

$$\begin{cases} shift(m,\, (bs \,@\, mkeps(r_1) \,@\, mkfin(r_1)),\, c,\, r_2) & \text{if } m \wedge nullable(r_1) \wedge fin(r_1) \\ shift(m,\, bs \,@\, mkeps(r_1),\, c,\, r_2) & \text{if } m \wedge nullable(r_1) \\ shift(\text{true},\, mkfin(r_1),\, c,\, r_2) & \text{if } fin(r_1) \\ shift(\text{false},\, [\,],\, c,\, r_2) & \text{otherwise} \end{cases}$$

$$shift(m, bs, c, r^*) \quad \overset{\text{def}}{=}$$

$$\begin{cases} \big(shift(m,\, (bs \,@\, mkfin(r)) \oplus 0,\, c,\, r)\big)^* & \text{if } m \wedge fin(r) \\ \big(shift(\text{true},\, mkfin(r) \oplus 0,\, c,\, r)\big)^* & \text{if } fin(r) \\ \big(shift(m,\, bs \oplus 0,\, c,\, r)\big)^* & \text{if } m \\ \big(shift(\text{false},\, [\,],\, c,\, r)\big)^* & \text{otherwise} \end{cases}$$

---

Figure 22: Bit-Annotated Version 2 *shift* . Here, $\oplus$ stands for appending a single element to each member of a list, $\lessdot$ stands for appending a list to each member of a list, and @ stands for list concatenation.

This reordering, which we call *reshuffle*, is applied after shifting through the first part of a sequence, after shifting into the inner regular expression inside a Star, and in the handling of $r^n$. When shifting is complete, we extract the bit sequences of the first mark whose carried suffix is empty (i.e., fully consumed).

We define *shifts* in this version in Figure 24. The behaviour of each case of the *shifts* function is discussed in detail below.

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

$$mkfin(\bullet_{bs} c) \quad \overset{\text{def}}{=} \quad bs$$

$$mkfin(r_1 + r_2) \quad \overset{\text{def}}{=} \quad \begin{cases} mkfin(r_1) \ @ \ mkfin(r_2) & \text{if } fin(r_1) \wedge fin(r_2), \\ mkfin(r_1) & \text{if } fin(r_1), \\ mkfin(r_2) & \text{if } fin(r_2). \end{cases}$$

$$mkfin(r_1 \cdot r_2) \quad \overset{\text{def}}{=}$$

$$\begin{cases} mkfin(r_2) \ @ \ \big(mkfin(r_1) \lessdot mkeps(r_2)\big) & \text{if } fin(r_1) \wedge nullable(r_2) \wedge fin(r_2), \\ mkfin(r_1) \lessdot mkeps(r_2) & \text{if } fin(r_1) \wedge nullable(r_2), \\ mkfin(r_2) & \text{otherwise.} \end{cases}$$

$$mkfin(r^*) \quad \overset{\text{def}}{=} \quad mkfin(r) \bigoplus 1$$

Figure 23: Bit-Annotated Version 2 *mkfin* . Returns a list of all bitsequences in a point, representing all possible match paths.

- Similar to the previous version described in Section 3.1.

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

- The list of marks $ms$ is shifted into both subexpressions $r_1$ and $r_2$, with 0 appended to each mark's bitsequence in the left-hand side $r_1$, and 1 appended in $r_2$.

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

- As in previous versions, this is the most elaborate case, especially with the ordering, which determines how marks are shifted and combined to preserve the remaining suffixes order. It begins by shifting the marks into the first part of the sequence, $r_1$, similar to the previous version, producing a new list of marks $ms'$. We then apply *reshuffle*, which reorders the list of marks based on the remaining suffixes, giving lower priority to those with longer remaining strings. The result then depends on several conditions, namely:

1. *nullable*($r_1$) $\wedge$ *nullable*($r_2$): when both $r_1$ and $r_2$ can be skipped, the returned list of marks must preserve the remaining suffixes order. There are three ordered parts in the result:

   (a) $r_1$ consumes and $r_2$ is skipped: $ms'$ is appended with $mkeps(r_2)$ to account for the case where $r_1$ consumes part of the string while $r_2$ is skipped.

   (b) $r_1$ and $r_2$ both consume: each mark in $ms'$ is shifted into $r_2$, accounting for both parts consuming characters. (The reason why each mark is shifted individually is explained in the final case below.)

   (c) $r_1$ is skipped and $r_2$ consumes: the original marks $ms$ are appended with $mkeps(r_1)$ and shifted into $r_2$, accounting for the case where $r_1$ is entirely skipped.

2. *nullable*($r_1$): if $r_1$ can be skipped, each mark of $ms'$ is first shifted into $r_2$, and then the original mark list $ms$, appended with $mkeps(r_1)$, is also shifted into $r_2$ to account for skipping $r_1$. This preserves the remaining suffixes order by prioritizing marks where $r_1$ consumes input over those where it is skipped, similar to case (3) above.

3. *nullable*($r_2$): if $r_2$ can be skipped, $ms'$ is first appended with $mkeps(r_2)$ (accounting for skipping $r_2$ entirely), and then each mark of $ms'$ is shifted into $r_2$. This preserves the remaining suffixes order by prioritizing marks where $r_1$ consumes while $r_2$ is skipped over those where both consume.

4. $\neg$*nullable*($r_1$) $\wedge$ $\neg$*nullable*($r_2$): if neither $r_1$ nor $r_2$ can be skipped, we shift each mark in $ms'$ individually into $r_2$. If we move both marks at the same time to the second part, we will lose the remaining suffixes order in cases where we have a list of marks from the first part that does not match all subexpressions in the second part, which would disrupt the correct ordering.

   Consider the example $((a + b) + ab) \cdot (bc + (c + b))$ with the string `abc`:

   (a) $\big|_{[\bullet\, abc]} ((a + b) + ab) \cdot (bc + (c + b))$

   (b) $((a + b) + ab) \big|_{[\bullet\, c,\, \bullet\, bc]} \cdot (bc + (c + b))$

   Here, $\bullet\, c$ has consumed the leftmost match via $ab$, while $\bullet\, bc$ consumed only $a$. If both marks were moved simultaneously, $\bullet\, c$ would

be dropped by the left subexpression since it does not match $bc$, and $\bullet\,bc$—which matches successfully—would produce $\bullet\,[\,]$ and take priority in the result of the alternative $r_2$, since the left and right mark lists are concatenated in order. To avoid this issue, we shift each mark individually into $r_2$: first, $\bullet\,c$ is shifted, producing an empty list for the left subexpression and $\bullet\,[\,]$ for the right; then $\bullet\,bc$ is shifted, producing $\bullet\,c$ for the left and $\bullet\,[\,]$ for the right. Combining both results keeps the remaining suffixes order and correctly prioritizes the POSIX-preferred value.

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

- Similar to the previous version, but with annotating 0 to indicate the beginning of a star iteration, and applying *reshuffle* to reorder the resulting list and preserve the remaining suffixes order. The end of an iteration is indicated by appending 1 when returning the marks representing that iteration.

**n-Repetitions case** $(r^n)$**:** The shifting depends on several cases depending on $n$.

- If $n = 0$, then no marks are shifted and an empty list is returned, keeping to the choice made in the previous versions for the empty string expression $\mathbf{1}$, (see A.1), since $n = 0$ represents the same language as $\mathbf{1}$.

- If $n = 1$, then we only shift the marks into the inner $r$, representing one iteration. The 0 is appended to indicate the beginning of the iteration, and we then apply *reshuffle* to preserve the remaining suffixes order.

- If $n > 1$, we first shift the marks into the inner $r$ with 0 appended to indicate the beginning of the iteration, and then apply *reshuffle* to preserve the remaining suffixes order. If the result is empty, we return an empty list. Otherwise, we have two cases:

  - If $r$ is nullable, we append 1 to indicate the end of the iteration without completing the full $n$ times due to skipping $r$, and shift the result of shifting into $r$ into $r^{n-1}$ to represent the remaining iterations. This accounts for the possibility of skipping $r$, similar to the behaviour in the star case.

39

– If $r$ is not nullable, we simply shift the result into $r^{n-1}$ without appending 1.

$$shifts(ms, 0) \quad \overset{\text{def}}{=} \quad [\,]$$

$$shifts(ms, 1) \quad \overset{\text{def}}{=} \quad [\,]$$

$$shifts(ms, d) \quad \overset{\text{def}}{=} \quad [\, \bullet\, s \mid \bullet\, d :: s \in ms \,]$$

$$shifts(ms, r_1 + r_2) \quad \overset{\text{def}}{=} \quad shifts(ms \oplus 0, r_1) \,@\, shifts(ms \oplus 1, r_2)$$

$$shifts(ms, r_1 \cdot r_2) \quad \overset{\text{def}}{=} \quad \text{let } ms' = shifts(ms, r_1).reshuffle \text{ in}$$

$$\begin{cases} (\, ms' \lessdot mkeps(r_2)\,) \,@\, \big[\, shifts([\bullet\, s], r_2) \mid \bullet\, s \in ms' \,\big]^{@} \,@\, shifts(ms \lessdot mkeps(r_1), r_2) \\ \text{if } nullable(r_1) \wedge nullable(r_2), \\ \big[\, shifts([\bullet\, s], r_2) \mid \bullet\, s \in ms' \,\big]^{@} \,@\, shifts(ms \lessdot mkeps(r_1), r_2) \\ \text{if } nullable(r_1) \wedge \neg nullable(r_2), \\ (\, ms' \lessdot mkeps(r_2)\,) \,@\, \big[\, shifts([\bullet\, s], r_2) \mid \bullet\, s \in ms' \,\big]^{@} \\ \text{if } \neg nullable(r_1) \wedge nullable(r_2), \\ \big[\, shifts([\bullet\, s], r_2) \mid \bullet\, s \in ms' \,\big]^{@} \\ \text{otherwise.} \end{cases}$$

$$shifts(ms, r^*) \quad \overset{\text{def}}{=} \quad \text{let } ms' = shifts(ms \oplus 0, r).reshuffle \text{ in}$$

$$\text{if } ms' = [\,] \text{ then } [\,] \text{ else } (\, ms' \oplus 1\,) \,@\, shifts(ms', r^*)$$

$$shifts(ms, r^n) \quad \overset{\text{def}}{=} \quad \text{if } n = 0 \text{ then } [\,]$$

$$\text{else if } n = 1 \text{ then } shifts(ms \oplus 0, r).reshuffle$$

$$\text{else let } ms' = shifts(ms \oplus 0, r).reshuffle \text{ in}$$

$$\text{if } ms' = [\,] \text{ then } [\,]$$

$$\text{else } \begin{cases} (\, ms' \oplus 1\,) \,@\, shifts(ms', r^{n-1}) & \text{if } nullable(r), \\ shifts(ms', r^{n-1}) & \text{otherwise.} \end{cases}$$

---

Figure 24: String-Carrying Marks - Lexer *shifts* function. Here, $\oplus$ stands for appending a single element to each member of a list, $\lessdot$ stands for appending a list to each member of a list, and @ stands for list concatenation.