Progress Report - 9 Months

Meshal Binnasban

August 10, 2025

This is the nine-month progress report for my PhD at King's College London. It summarises my work over the period—motivation, challenges, and the development and refinement of a marked-based regular expression matcher for POSIX disambiguation.

Project Synopsis

This research investigates the use of the marked approach for regular expression matching, aiming to address limitations found in derivative-based methods—particularly the exponential growth of intermediate expressions, which makes them difficult to scale in practice. The work explores how the propagation of marks through regular expressions can be used to track matching progress and construct parse values.

Bitcoded representations are explored to record the construction of values and guide the selection of the POSIX-preferred parse. Particular focus is placed on handling complex constructs such as nested iteration (STAR) and bounded repetition (NTIMES), with future work aiming to explore additional operators such as intersection and negation.

While Brzozowski derivatives can extract POSIX values (as demonstrated by Sulzmann and Lu [7]), they suffer from size explosion as noted earlier. Partial derivatives mitigate some of this growth but can still expand cubically and currently lack support for full value extraction. The marked approach aims to leverage the fact that marks propagate without generating new subexpressions, enabling better control over disambiguation while maintaining efficiency.

The design is validated through comparisons with a derivative-based reference matcher, using large-scale testing to uncover edge cases and confirm

correctness. Although current versions use bitcodes to annotate match structure, the underlying method is adaptable to other representations.

The long-term aim is to develop a matching algorithm that consistently yields POSIX-disambiguated values for all regular expressions and to formally verify its correctness, with potential extensions to symbolic or automata-theoretic frameworks.

1 Derivatives

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

The notion of derivatives in regular expressions is well established, but have gained attention in the last decade [6, 4]. Their simplicity and compatibility with functional programming have renewed interest in their use [3].

To decide whether a string $w = a_1 a_2 \ldots a_n$ matches a regular expression r using derivatives, we iteratively compute:

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

Then test whether r_n can match the empty string.

To illustrate how derivatives can be used to match a regex against a string, consider the regular expression $(ab + ba)$. The derivative can check the matching of a string like `ba` by taking the derivative of the regex with respect to `b`, then `a`.

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

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

Since the remaining derivative now matches the empty string and there are no more characters left, the string `s = ba` matches the regex.

Some subexpressions in the derivatives may be unnecessary. For example, $(0 \cdot b)$ can understandably be simplified to 0, since 0 denotes the empty language. this might be necessary, because the derivative method, while elegant and recursive in structure, suffers in practice from duplicating large portions of the regex tree and sensitivity to syntactic form, requiring simplifications

Applying simplifications under ACI rules (associativity, commutativity, and idempotence) as well as rules concerning the empty string and empty

language, e.g. $(0 \cdot r = 0\ )$ provides a finite bound on the number of intermediate expressions produced by derivatives [8]. These simplifications help mitigate the explosion in the size of derivatives, a well-known issue as noted by Sulzmann and Lu [7] and further discussed in Section 1.2.

1.1 Derivative Extension

Sulzmann and Lu extend Brzozowski's derivatives by offering lexing information in addition to matching by embeding bitcode annotations—representing parse trees—into the regular expressions during derivative construction. This bitcode is extracted after matching and then decoded back into a parse tree, yielding the POSIX-compliant parse value.

Bitcodes are lists of 0s and 1s that record the path taken to reach a match. These bits encode choices in `ALT`, repetitions in `STAR` during matching.

The following is an example of how bitcoding works in regular expressions.

$$'ab' \rightarrow (a + ab)(b + \epsilon)$$

We proceed by constructing bitcoded derivatives step-by-step, and tracking how the bitcode grows:

1. Step 1: Intern the regex

$$r' = \ intern(r) = \ \ ({}_0a +_1 ab) \cdot ({}_0b +_1 \epsilon)$$

2. Step 2: input a

$$\begin{array}{rcl} der_a(r') & = & der_a(({}_0a +_1 ab) \cdot ({}_0b +_1 \epsilon)) \\ & = & der_a(({}_0a +_1 ab)) \cdot ({}_0b +_1 \epsilon) \\ & = & (der_a({}_0a) + der_a({}_1ab)) \cdot ({}_0b +_1 \epsilon) \\ & = & ({}_01 + der_a({}_1a) \cdot b) \cdot ({}_0b +_1 \epsilon) \\ & \rightarrow & ({}_01 +_1 1 \cdot b) \cdot ({}_0b +_1 \epsilon) \end{array}$$

3. Step 3: input b

$$\begin{array}{rcl} der_b(der_a(r')) & = & der_b(({}_01 +_1 1 \cdot b) \cdot ({}_0b +_1 \epsilon)) \\ & = & der_b(({}_01 +_1 1 \cdot b)) \cdot ({}_0b +_1 \epsilon) + \ der_b(fuse(mkeps(r1), ({}_0b +_1 \epsilon))) \\ & = & (der_b({}_01) + der_b({}_11 \cdot b)) \cdot ({}_0b +_1 \epsilon) + \ ({}_0(der_b({}_0b) + der_b({}_1\epsilon))) \\ & = & (0 + der_b({}_11 \cdot b)) \cdot ({}_0b +_1 \epsilon) + \ ({}_0({}_01 + 0)) \\ & = & ((der_b({}_11) \cdot b) + der_b({}_1b)) \cdot ({}_0b +_1 \epsilon) + \ ({}_0({}_01 + 0)) \\ & \rightarrow & ((0 \cdot b) +_1 1) \cdot ({}_0b +_1 \epsilon) + \ ({}_0({}_01 + 0)) \end{array}$$

Initially, the algorithm will intern the regular expression to prepare it for processing. In this step, the left subexpressions are fused with bitcode ‘0‘s, and the right ones with ‘1‘s [7]. Then, it takes the derivative with respect to the first character of the input string, which in this case is ‘’a’‘.

The result of step 2 is that the ONEs/epsilons indicate a match, and the bitcode encodes how that match was obtained. So far, the first match can be obtained by going left to match `'a'`, which is also reflected in the bitcode `0`. The second match is found by going right to match the `'a'` in the sequence `'ab'`. The function `mkeps`, as defined by Sulzmann and Lu [7], extracts the bitcode from nullable derivatives. Its definition will be given at the end of this section.

The resulting derivative from step 2 expands into an alternative in step 3. This happens because the concatenation has become nullable, which means the first part, r_1, could be skipped while matching. To account for this, the derivative expands into an alternative: the left branch assumes r_1 wasn't skipped, and the right branch assumes it was, taking the derivative of r_1 directly. This is one of the reasons why derivatives tend to grow in size. Sulzmann and Lu use $fuse(mkeps(r_1))$ [7] to include the bits needed in case r_1 is skipped. These bits, extracted by `mkeps`, indicate how r_1 matched the empty string.

After taking the derivative with respect to the input string, the algorithm checks the nullability of the result. If it is nullable, it then calls `mkeps` to extract the bitcode that indicates how the match was obtained. In this example, there are two possible matches, but the algorithm prefers the left one. As a result, `mkeps` returns the bitcode `[1,1]`, which indicates that, in the first alternative of r_1 in the concatenation, it chose the right branch matching `'ab'`. Then, for the second part of the concatenation, it again chose the right branch, indicating a match with the empty string.

Final bitcode after calling `mkeps` on the resulting derivative: $[1, 1]$. Decoding this with the original expression yields the POSIX parse tree:

```
Seq(Right(Seq('a', 'b')), Right(Empty))
```

******add functions definitions; der , mkeps, fuse, intern******

1.2 Size Explosion.

Even with aggressive simplifications—as shown by Sulzmann and another variant of the algorithm in *POSIX Lexing with Bitcoded Derivatives* [8]—

the simplifications keep the number of derivatives finitely bounded, the number can still grow significantly, making practical usage difficult. The size of derivatives can arise to seemingly infinite proportions even for some simple expressions. Consider the case $(a + b)^* \cdot c$ with input `aaac`. The number of expressions can grow significantly because each derivative may introduce new structure. Each step unfolds all possible ways the `STAR` expression can match the input. Urban and Tan showed that even under simplifications such as removing redundant subterms and collapsing identical alternatives, the size remains only finitely bounded, but still grows quickly. Even Antimirov's partial derivatives [1], which do not track POSIX parse values, may produce cubic growth in worst cases.

2 Marked Approach

The marked approach is a method for regular expression matching that tracks the progress of matching by inserting marks into the regular expression. As noted by Nipkow and Traytel [5], the concept of marks dates back to earlier works, but recent work by Fischer [3] and a variation by Asperti [2] developed the idea further in a modern form.

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

There is a slight difference in the interpretation of marks in the works of Fischer [3] and Asperti [2]. In Fischer's work, marks are inserted after a character has been matched, indicating a matched character or subexpression. In contrast, Asperti interprets the positions of the marks as indicating the ability to match a character. With each character consumed, the marks are moved through the regex into new positions that indicate the next character that can be consumed. Similar to Fischer's work, at the end, the regex is evaluated to determine whether it is in a final state; in Asperti's case, this means that the marks are in positions where the regex can accept the empty string, which reflects that all input characters have been consumed. If the marks were not in those positions, it would mean that some characters were not properly matched and the expression should not be accepted.

2.1 Motivation for a Marked Approach

The marked approach offers an alternative to derivatives for regular expression matching. It depends on *moving/shifting* markers around the regex instead of creating new subexpressions. Our main motivation is that this method could potentially support fast and high-performance POSIX matching and value extraction, based on how it handles matching and avoids the limitations found in the derivatives approach.

Inspired by the works of Fischer et al. and Asperti et al. [3, 2], we aim to leverage the marked approach and extend it to handle POSIX-compliant disambiguation, as well as complex constructors used in modern regexes, such as bounded repetitions and intersections. The work we are doing is mostly inspired by the algorithm described by Fischer [3]. As shown by Nipkow and Traytel [5], the pre-mark algorithm (from Asperti) is actually a special case of the post-mark algorithm (from Fischer), so Fischer's version appears to be the most suitable base for extending to POSIX value disambiguation.

2.2 Versions and Implementation

We began our work by implementing the marked approach as described by Fischer [3] in the Scala language. 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 is carried out by a function called *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.

Scala Implementation of Fischer's Marked Approach

The following describes the shifting behavior as defined by Fischer [3].

$$shift(m, c, r) \rightarrow Marked\,r$$

The three arguments are: the regular expression to match against, a boolean indicating whether to introduce a new mark or just move existing ones, and the input character. The boolean determines the mode of operation: when set to `true`, a new mark is introduced; otherwise, the function shifts the existing marks. In our implementation, we added a boolean attribute to the

`CHAR` constructor to represent a marked character. In later versions, we use a wrapper constructor around `CHAR` to explicitly represent a marked character.

$$
\begin{array}{rcl}
shift(m, c, \mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
shift(m, c, \mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{1} \\
shift(m, c, d) & \stackrel{\text{def}}{=} & \begin{cases} \bullet d & \text{if } c = d \wedge m \\ d & \text{otherwise} \end{cases} \\
shift(m, c, r_1 + r_2) & \stackrel{\text{def}}{=} & shift(m, c, r1) + shift(m, c, r2) \\
shift(m, c, r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \begin{cases} shift(m, c, r_1) \cdot shift(true, c, r_2) & \text{if } m \wedge nullable\ r_1 \\ shift(m, c, r_1) \cdot shift(true, c, r_2) & \text{if } fin(r_1) \\ shift(m, c, r_1) \cdot shift(false, c, r_2) & \text{otherwise} \end{cases} \\
shift(m, c, r^*) & \stackrel{\text{def}}{=} & \begin{cases} shift(m, c, r^*) & \\ shift(true, c, r^*) & \text{if } fin(r) \end{cases}
\end{array}
$$

Shifting marks for the base cases `ZERO` and `ONE` is straightforward, as they cannot have any marks on them.

- **Character case** (d): A mark is set if the character matches the input character and the mode is `true`. The mode controls how marks propagate through the regular expression.
- **Alternative case** ($r_1 + r_2$): Marks are shifted into both branches, since either branch could match the same input character.
- **Sequence case** ($r_1 \cdot r_2$):
  - If neither r_1 is nullable nor in a final position, shift only into r_1, indicating that we are matching the first part of the sequence.
  - If r_1 is nullable and can be skipped, also shift into r_2 so that both parts can begin matching.

  - If $fin(r_1)$ holds (meaning r_1 has finished matching), shift into r_2 to begin matching its part.

- **Star case** (r^*): Shift into the subexpression if the mode is `true` or if the subexpression is in a final position.

Helper Functions Definitions:

- $fin(regex) \rightarrow Boolean$

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

- $nullable(regex) \rightarrow Boolean$

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

This first implementation of the algorithm, provides only acceptance checking without any value construction, the followingsubsections describe the different versions we have developed so far.

2.2.1 Bit-Annotated POINT, version 1

In this version, we extended the marked approach to include bitcodes that annotate the marks that is being shifted through the regex. inspired by Sulzmann and Lu [7], we introduced a bitcode in form of a list that is attached to each mark, and it gets built as the marks are shifted.

This version produces a value but not necessarily the POSIX-preferred value. We use the bit annotations Z (or 0) and S (or 1), similar to the bitcoded derivative. In this version, the function $shift$ takes an additional argument, $Bits$, which is a list of Bit elements. When we shift through this function, bits are added to the list. If a mark is to be added to a leaf character node, this list is stored in the `POINT` constructor that wraps the marked character constructor. We define two additional functions: `mkfin`, which extracts the bit sequence of a final constructor (that is, the path, in bits, describing how the expression matched), and `mkeps`, which extracts the bit sequence of a nullable expression matching the empty string (showing the path that led to the empty-string match). The definitions are given below, starting with the $shift$ function.

$$shift(m, bs, c, 0) \stackrel{\text{def}}{=} 0$$

$$shift(m, bs, c, 1) \stackrel{\text{def}}{=} 1$$

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

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

$$shift(m, bs, c, r_1 \cdot r_2) \stackrel{\text{def}}{=} \begin{cases} shift(m, bs, c, r_1) \cdot shift(true, bs +\!\!+ mkeps(r_1), c, r_2) & \text{if } m \wedge \\ shift(m, bs, c, r_1) \cdot shift(true, mkfin(r_1), c, r_2) & \text{if } fin(\\ shift(m, bs, c, r_1) \cdot shift(false, [\,], c, r_2) & \text{otherw} \end{cases}$$

$$shift(m, bs, c, (r)^*) \stackrel{\text{def}}{=} \begin{cases} (shift(m, bs \oplus Z, c, r))^* & \text{if } m \\ (shift(true, bs +\!\!+ (mkfin(r) \oplus S), c, r))^* & \text{if } m \wedge fin(r) \\ (shift(true, mkfin(r) \oplus Z, c, r))^* & \text{if } fin(r) \\ (shift(false, [\,], c, r))^* & \text{otherwise} \end{cases}$$

The $\oplus$ denotes adding a bit at the end of a bit list, and $+\!\!+$ denotes the concatenation of two bit lists. The symbols Z (or 0) and S (or 1) are used to represent left and right choices, respectively.

Alternative case $(r_1 + r_2)$:

- We shift as before and annotate the direction of the match with Z or S, adding this bit to the list of bits.

Sequence case $(r_1 \cdot r_2)$:

- If r_1 is nullable, we shift a mark to both r_1 and r_2, passing bs to r_1 (representing the current path leading to this expression) and $bs + +\,\texttt{mkeps}$ to r_2, where `mkeps` returns the bits for an empty-string match. This corresponds to the path for this sequence when the first part is skipped.

- If r_1 is in a final position (meaning it has finished matching), we pass a mark to r_2 with the bit list describing how r_1 was matched, extracted using the `mkfin` function.

- Otherwise, we pass a new mark to r_1 only, with bs representing how we reached r_1 in this sequence.

Star case (r^*):

- If only we are introducing a new mark, we pass bs, representing the bits that describe how the r^* expression was reached and append Z to indicate a new iteration has begun.

- If we are introducing a new mark and r is in a final position, we pass $bs + +\,\texttt{mkfin}(r)$, combining the bits representing the path to r^* with the bits from `mkfin` that describe how r reached its final position with S representing an end of one iteration. So to say, to append the bitcode for how r reaches a final position to the bitcode of the new mark to be introduced to the STAR.

- If r is in a final position, then we pass mkfin which describe how r reached its final position and then append Z representing the begining of a new iteration.

- If no new mark is introduced, the existing ones are moved by passing an empty list of bits.

Next, we present two examples of matching a string and extracting a value. The first example demonstrates how the bit sequence is constructed during matching, while the second illustrates why the algorithm does not always produce the POSIX-preferred value. In version 1, when shifting to a point (an already marked character) and the character matches again, the associated bit list is overwritten. This behaviour can cause value erasure

and, in certain cases, the loss of the POSIX-preferred value, as will be shown in the second example.

1. **Example 1:** $'ba' \rightarrow (a \cdot b + b \cdot a)$

$$shift\ b \ \rightarrow \ (a \cdot b) + (\ _{\{S\}} \bullet b \cdot a)$$

$$shift\ a \ \rightarrow \ (a \cdot b) + (b \cdot\ _{\{S\}} \bullet a)$$

   With no further calls to `shift`, `mkfin` is called because the regular expression has reached a final position, indicated by a mark at the end of the right-hand subexpression in the alternative. `mkfin` then retrieves the bit list $\{S, S\}$.

2. **Example 2:** $'aaa' \rightarrow (a + a \cdot a)^*$

$$shift\ a \ \rightarrow \ (_{\{Z,Z\}}\bullet a +\ _{\{Z,S\}} \bullet a \cdot a)^*$$

$$shift\ a \ \rightarrow \ (_{\{Z,Z,Z,Z\}}\bullet a +\ _{\{Z,Z,Z,S\}} \bullet a \cdot\ _{\{Z,S\}} \bullet a)^*$$

$$shift\ a \ \rightarrow \ (_{\{Z,Z,Z,Z,Z,Z\}}\bullet a +\ _{\{Z,Z,Z,Z,Z,S\}} \bullet a \cdot\ _{\{Z,Z,Z,S\}} \bullet a)$$

   In this example, after the first shift on a, a mark is placed on the left branch with bits $\{Z, Z\}$, indicating the start of a `STAR` iteration followed by a left choice. In the right subexpression, the mark on r_1 of the ab sequence carries bits $\{Z, S\}$, representing the start of the `STAR` iteration followed by a right choice.

   By the third shift, the bits in the right subexpression $\{Z, S\}$ are overwritten, when they should instead be preserved. These bits correspond to the POSIX-preferred match, which starts by matching the right-hand side first, then performing another iteration to match the left-hand side. The correct bit sequence in that case would be $\{Z, S, Z, Z\}$, with the final S marking the end of the `STAR` iteration. This behaviour arises because the `POINT` wrapper stores only a single bit list at a time, and in this version there is no clearly defined ordering of marks.

The annotation can become difficult to follow, which motivated us to introduce additional symbols forming sequences (transitioning from a bit-based representation to sequences) in later versions. For instance, we use N to denote the beginning of a `STAR` iteration (replacing Z) and E to denote the end of an iteration (replacing S).

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

2.2.2 Bit-Annotated POINT, version 2

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

2.2.3 Input-Carrying Marks

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

*** from previous report ***

3 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-preferred value is selected in all cases involving repetition remains a primary target. The current implementation explores candidate paths, but the disambiguation logic for selecting among them requires refinement and formal confirmation.
- **Support for Additional Operators.** Beyond the basic constructs (ALT, SEQ, STAR, NTIMES), future work includes extending the matcher to handle additional regex operators such as intersection, negation, and lookahead. These additions require careful definition of how

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

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

References

[1] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[2] A. Asperti, C. S. Coen, and E. Tassi. Regular Expressions, au point. *arXiv, http://arxiv.org/abs/1010.2604*, 2010.

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

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

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