Progress Report - 9 Months

Meshal Binnasban

July 31, 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.

Outline

1. Revisiting Derivatives
2. The Problem of Size Explosion
3. Attempts at Simplification
4. Motivation for a Marked Approach
5. The Marked Approach: Versions and Implementation

1 Revisiting Derivatives

The idea of derivatives of regular expressions was introduced by Brzozowski [3] in the 1960s. The method allows constructing a recogniser for a regular language by successively taking the derivative of a regular expression with respect to each character in the input string. Derivatives have reappeared in functional programming contexts, notably in work by Owens et al. [6] and by Might et al. [5].

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

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

and then test whether r_n can match the empty string.

While elegant and recursive in structure, the derivative method suffers in practice from:

- Duplicating large portions of the regex tree
- Sensitivity to syntactic form, requiring simplifications

2 The Problem of Size Explosion

A central challenge is the rapid growth in derivative size. For instance, consider the regex $(a+b)^* \cdot c$ and the input string $aaac$. Repeated derivatives such as:

$$d_a(d_a(d_a(d_a((a + b)^* \cdot c))))$$

yield deeply nested alternations and STAR expressions.

In practice, this results in:

- Memory blowup
- Redundant recomputation
- Exponential size of intermediate expressions

This makes real-time matching and lexing infeasible without aggressive simplification.

3 Attempts at Simplification

To reduce derivative complexity, common algebraic simplifications are applied after each derivative step [3].

- $r + \emptyset = r$
- $\emptyset + r = r$
- $r \cdot \varepsilon = r$
- $\varepsilon \cdot r = r$
- $r + r = r$

However, these simplifications are often insufficient to prevent growth in practice, especially in the presence of nested constructs like alternation and iteration. Consider the expression

$$((a + b)^* \cdot a) + a$$

and the input `aac`. After the first derivative step with respect to the first `a`, we get:

$$der_a(r) = ((a + b)^* \cdot a) + \epsilon$$

The second derivative, again with respect to `a`, unfolds the STAR again, producing:

$$der_a(der_a(r)) = (((a + b)^* \cdot a) + \epsilon) + \ldots$$

Even though simplification rules are applied at each stage, the repeated unfolding of $(a + b)^*$ causes structurally distinct expressions to proliferate.

This behaviour is typical of derivative-based matching, where constructs such as STAR repeatedly unfold into new expressions. Since syntactic identity (required by simplification rules like $r + r = r$) is not always preserved during unfolding, structurally different but semantically equivalent expressions may accumulate, preventing simplification from taking effect.

This growth is not unique to Brzozowski's formulation. In his 1996 work, Antimirov shows that the number of partial derivatives is bounded by $(size(r) + 1)^3$ [1]. While this bound is polynomial, it still allows for hundreds or thousands of distinct expressions to arise in practical cases. Thus, even partial derivatives—despite being more concise than full derivatives—may still unfold in ways that make it hard to scale the matcher effectively.

4 Motivation for a Marked Approach

While derivatives can accept or reject strings, they do not provide parse values or disambiguation information. For tasks like POSIX lexing, this is a major limitation.

Inspired by the work of Fischer et al. and Asperti et al. [4, 2], the marked approach offers an alternative. It inserts *points* into regexes to represent where in the expression the matching is occurring.

Our aim is to extend the marked approach to:

- Extract full match values
- Represent all possible parse trees via bitcodes
- Support POSIX-compliant disambiguation
- Handle complex constructs like STAR and NTIMES

5 Marked Approach: Versions and Implementation

Version 1: Basic Matching with POINT

A `POINT` wraps a character inside the regex. The shift function moves the point through the expression.

Key ideas:

- If POINT reaches a matching character, we record it
- In ALT $(r_1 + r_2)$, the point is passed to both sides
- In SEQ $(r_1 \cdot r_2)$, it moves through r_1; if nullable, also into r_2
- STAR recursively applies the same logic

Example: Matching `aa` against $(a + b) \cdot a$
The point travels first to $a + b$ (left side), then to the final a. The matching path is encoded as bitcode $[0, 1]$ (Left, then Right).

Limitations: Only one mark is preserved at a time, so alternative parses are lost.

Version 2: Bit-Annotated POINT

Each POINT now stores a bitcode list during propagation. This helps trace the matching path.

Example: Matching `aa` against $(a + a) \cdot a$
Two valid parses: one going left, one going right. Bitcodes like $[0, 1]$ and $[1, 1]$ distinguish these. However, we still overwrite older points during propagation.

Problem: We still lose some values due to destructive replacement.

Version 3: Collecting All Bit Paths

We now retain *all* possible POINTs, each with their own bitcode. The final matcher returns a list of bitcode sequences representing all valid parses.

Example: Matching `aac` against $((a + b)^* \cdot a) + a$ yields:

- STAR + SEQ path: $[N, L, L, E]$
- ALT shortcut path: $[R]$

Testing over 10 billion cases confirms the POSIX-correct value is always in the output.

5.1 Version 4: Input-Carrying Marks

In this version, each mark holds its own input string. Initially, we have a single active mark with the full input. Shifting consumes the input and returns new marks that reflect the updated state of both the regex and the remaining input.

Example: Regex: $a^* \cdot b$, Input: *aaab* Marks evolve as: `Mark(true, "aaab")` → `Mark(true, "aab")` → `Mark(true, "ab")` → `Mark(true, "b")` → `Mark(true, "")`

A match is considered successful if any mark reaches an empty input string and the corresponding regex is in final state.

Current Status: Most constructs (e.g., ALT, SEQ, NTIMES) currently work as intended. However, POSIX-disambiguated matching for nested or ambiguous `STAR` expressions is still under development. The correct disambiguation logic for STAR—particularly in the presence of nullable subexpressions or nested repetition—remains an ongoing area of active implementation and testing.

Version 4: Input-Carrying Marks

In this version, each mark is a tuple `(str, bits)`, where `str` is the remaining input. and `bits` is the accumelate bit sequence representing the path taken by the mark through the regex. The `shift` function consumes characters from `str` and outputs a new list of marks.

Example:

Regex: $a^*.b$, Input:$aaab$ Marks evolve as:

`(true, "aaab")` → `(true, "aab")` → `(true, "ab")` → `(true, "b")` → `(true, "")`

This version prioritize marks by remaining string.

Key Functions

- `shift(m, a, r)`: advance marks through regex
- `mkfin(r)`: extract final POINTs and bitcodes
- `mkeps(r)`: compute nullable parse values

6 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] 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] 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.