Progress Report - 9 Months

Meshal Binnasban

July 30, 2025

This report presents the progress achieved during the first nine months of my PhD research at King's College London. The research investigates the use of the marked approach for regular expression matching, aiming to develop an alternative to derivative-based algorithms while supporting full value extraction and 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, such as the exponential growth of derivatives that can slow down the matching process. The study focuses on developing and implementing the marked approach by writing key code and functions to control the movement of marks through regular expressions. Additionally, it explores bitcoded matching techniques, including the implementation of several functions to effectively track match paths and reconstruct parse trees. The research also examines disambiguation strategies to replicate POSIX behavior, tackling complex constructs like nested STAR operations and bounded repetitions (NTIMES). Comparative analyses with derivative-based matchers are conducted to identify counterexamples and refine the approach. The ultimate goal is to develop an efficient and reliable alternative to derivative-based matching algorithms that adheres to POSIX semantics.

1 Outline

1. Derivatives: old technology rediscovered
2. Exploding size problem
3. Simplification attempts (still examples where they explode)

4. Motivation: POSIX disambiguation and limitations of existing approaches

5. Marked approach with multiple versions

2 Derivatives

Derivatives of regular expressions were introduced by Brzozowski [2] and later applied in functional settings [4, **?**]. Given a regex r, the derivative $d_a(r)$ denotes a new regex that matches all suffixes of strings in $L(r)$ that start with the character a.

Example:
Given $r = a \cdot b$, we have:

$$d_a(a \cdot b) = d_a(a) \cdot b = \varepsilon \cdot b = b$$

Although simple and elegant, derivative computation can lead to an exponential number of intermediate expressions.

3 The Problem of Size Explosion

One core issue with derivatives is the uncontrolled growth in the number and size of expressions during matching. For instance, even a simple regex like $(a+b)^* \cdot c$ leads to many distinct expressions when derivatives are taken repeatedly.

Example:
Let $r = (a + b)^* \cdot c$, and consider the string $aaac$. The repeated derivatives:

$$d_a(d_a(d_a(d_a(r))))$$

involve nested alternations and sequences, and simplification becomes costly.

4 Simplification Attempts

To manage size, we apply simplifications such as:

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

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

However, even with these simplifications, the size can grow prohibitively. Some expressions generate deeply nested trees that require aggressive normalization strategies to keep manageable.

Example:
Regex: $((a + \varepsilon)^* + b)^* \cdot c$, with input $a^n c$ leads to exponential unfolding.

5 Motivation

While derivatives allow matching, they do not provide full parse trees or control over disambiguation. In lexing contexts—especially POSIX-compliant tools—returning the correct parse matters. Our goal is to build a matching engine that:

- Extracts full match values, not just accept/reject,
- Supports POSIX disambiguation,
- Tracks bitcoded paths for reconstructing parse trees.

6 The Marked Approach: Versions and Implementation

We follow the general idea introduced in [3] and [1], where matching is guided by inserting marks (or points) into the regex. The mark is used to track how the input is matched, and later extract a bitcode or parse value.

Version 1: Value Extraction with Points

In the first version, a single mark is inserted using a *POINT* constructor, wrapping a character. The shift function propagates this mark through the regex.

Shift behavior:

1. If the regex is a character and the mark is active and input matches: return a POINT.
2. If the regex is ALT $(r_1 + r_2)$, shift both sides.

3. If the regex is SEQ $(r_1 \cdot r_2)$, shift r_1 first. If r_1 is nullable, also shift into r_2.

4. If the regex is STAR, recursively shift into the body.

mkfin: Used to extract bitcode when part or all of the regex becomes final.

mkeps: Extracts how the empty string matches if the regex is nullable.

Example:

Regex: $(a + b) \cdot a$ with input aa

The POINT may follow one path, say Left then Right (bitcode $[0, 1]$), but some possible correct values are lost because some POINTs get overwritten during shifting.

Version 2: Propagating Bit Sequences in POINT

We extend POINT to store a bitcode list that accumulates during shifting.

Example:

Regex: $(a + a) \cdot a$ with input aa

Two parses possible: Left-Right or Right-Right. We propagate bits like $[L, R]$ or $[R, R]$ into the POINT. However, only one POINT is preserved (others get replaced), meaning we lose some values.

Key challenge: Mark replacement discards alternative parses that may be POSIX-preferred.

Version 3: Tracking All Paths

We change POINT to carry a list of bitcode lists. During shifting, all valid bitcode paths are preserved.

mkfin now returns a full list of all possible matching bitcodes.

mkeps extracts all nullability-based values.

Example:

Regex: $(a + b)^* \cdot c$, input: $aabc$

Bitcode paths like $[N, L, L, R, E]$ and $[N, L, R, E]$ are captured. This version passed validation against a derivative-based POSIX engine using over 10 billion tests.

We believe this version computes all possible parse paths that lead to acceptance. The POSIX value can then be selected using disambiguation.

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.

Example:

Regex: $a^* \cdot b$, Input: $aaab$

Marks carry: $aaab \to aab \to ab \to b \to \varepsilon$

A regex is final if one mark has consumed all characters.

Advantages:

- We can represent suspended marks for later resumption.
- It enables future features like controlling STAR behavior (e.g., delaying or preferring paths).
- Prioritization can be based on mark content, not just regex structure.

7 Conclusion and Future Work

This report presented four distinct versions of a marked regular expression matcher, each building toward full POSIX-correct value extraction. We validated version 3 against a reference engine and laid the foundation for value disambiguation in version 4.

Future plans:

- Refine disambiguation logic in version 4.
- Implement NTIMES and STAR disambiguation under POSIX.
- Compare run-time performance against derivatives.

References

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

[2] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, Oct. 1964.

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