# POSIX Lexing with Bitcoded Derivatives

**Chengsong Tan** ✉
King's College London

**Christian Urban** ✉
King's College London

## ─── Abstract ───

Sulzmann and Lu described a lexing algorithm that calculates Brzozowski derivatives using bit-sequences annotated to regular expressions. Their algorithm generates POSIX values which encode the information of *how* a regular expression matches a string—that is, which part of the string is matched by which part of the regular expression. The purpose of the bit-sequences in Sulzmann and Lu's algorithm is to keep the size of derivatives small which is achieved by 'aggressively' simplifying regular expressions. In this paper we describe a slight variant of Sulzmann and Lu's algorithm and *(i)* prove that this algorithm generates unique POSIX values; *(ii)* we also establish a cubic bound for the size of the derivatives—in earlier works, derivatives can grow exponentially even after simplification.

## 1 Introduction

Brzozowski [3] introduced the notion of the *derivative* $r \backslash c$ of a regular expression $r$ w.r.t. a character $c$, and showed that it gave a simple solution to the problem of matching a string $s$ with a regular expression $r$: if the derivative of $r$ w.r.t. (in succession) all the characters of the string matches the empty string, then $r$ matches $s$ (and *vice versa*). The derivative has the property (which may almost be regarded as its specification) that, for every string $s$ and regular expression $r$ and character $c$, one has $cs \in L\, r$ if and only if $s \in L\,(r \backslash c)$. The beauty of Brzozowski's derivatives is that they are neatly expressible in any functional language, and easily definable and reasoned about in theorem provers—the definitions just consist of inductive datatypes and simple recursive functions. A mechanised correctness proof of Brzozowski's matcher in for example HOL4 has been mentioned by Owens and Slind [9]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [6]. And another one in Coq is given by Coquand and Siles [4].

If a regular expression matches a string, then in general there is more than one way of how the string is matched. There are two commonly used disambiguation strategies to generate a unique answer: one is called GREEDY matching [5] and the other is POSIX matching [1, 7, 8, 10, 11]. For example consider the string $xy$ and the regular expression $(x + y + xy)^\star$. Either the string can be matched in two 'iterations' by the single letter-regular expressions $x$ and $y$, or directly in one iteration by $xy$. The first case corresponds to GREEDY matching, which first matches with the left-most symbol and only matches the next symbol in case of a mismatch (this is greedy in the sense of preferring instant gratification to delayed repletion). The second case is POSIX matching, which prefers the longest match.

## 2 Background

Sulzmann-Lu algorithm with inj. State that POSIX rules. metion slg is correct.
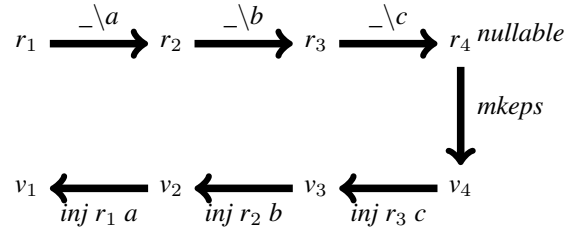
$$r_1 \xrightarrow{\_\backslash a} r_2 \xrightarrow{\_\backslash b} r_3 \xrightarrow{\_\backslash c} r_4 \ \textit{nullable}$$

$$\downarrow \textit{mkeps}$$

$$v_1 \xleftarrow[\textit{inj } r_1 \ a]{} v_2 \xleftarrow[\textit{inj } r_2 \ b]{} v_3 \xleftarrow[\textit{inj } r_3 \ c]{} v_4$$

**Figure 1** The two phases of the algorithm by Sulzmann & Lu [10], matching the string $[a, b, c]$. The first phase (the arrows from left to right) is Brzozowski's matcher building successive derivatives. If the last regular expression is *nullable*, then the functions of the second phase are called (the top-down and right-to-left arrows): first *mkeps* calculates a value $v_4$ witnessing how the empty string has been recognised by $r_4$. After that the function *inj* "injects back" the characters of the string into the values.

$$\frac{}{[] \in (\mathbf{1}) \to \textit{Void}}P\mathbf{1} \qquad \frac{}{[c] \in (c) \to \textit{val.Char } c}Pc$$

$$\frac{s \in r_1 \to v}{s \in (r_1 + r_2) \to \textit{val.Left } v}P{+}L \qquad \frac{s \in r_2 \to v \qquad s \notin L\,r_1}{s \in (r_1 + r_2) \to \textit{val.Right } v}P{+}R$$

$$\frac{\begin{array}{c} s_1 \in r_1 \to v_1 \qquad s_2 \in r_2 \to v_2 \\ \nexists s_3\, s_4.\, s_3 \neq [] \wedge s_3 \,@\, s_4 = s_2 \wedge s_1 \,@\, s_3 \in L\,r_1 \wedge s_4 \in L\,r_2 \end{array}}{(s_1 \,@\, s_2) \in (r_1 \cdot r_2) \to \textit{Seq } v_1\, v_2}PS$$

$$\frac{}{[] \in (r^\star) \to \textit{Stars } []}P[]$$

$$\frac{\begin{array}{c} s_1 \in r \to v \qquad s_2 \in (r^\star) \to \textit{Stars } vs \qquad \textit{flat } v \neq [] \\ \nexists s_3\, s_4.\, s_3 \neq [] \wedge s_3 \,@\, s_4 = s_2 \wedge s_1 \,@\, s_3 \in L\,r \wedge s_4 \in L\,(r^\star) \end{array}}{(s_1 \,@\, s_2) \in (r^\star) \to \textit{Stars } (v \cdot vs)}P\star$$

**Figure 2** Our inductive definition of POSIX values.

## 3 Bitcoded Derivatives

bitcoded regexes / decoding / bmkeps gets rid of the second phase (only single phase) correctness

## 4 Simplification

not direct correspondence with PDERs, because of example problem with retrieve
   correctness

## 5 Bound - NO

## 6 Bounded Regex / Not

## 7 Conclusion

[2]

### References

**1** The Open Group Base Specification Issue 6 IEEE Std 1003.1 2004 Edition, 2004. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html.

**2** F. Ausaf, R. Dyckhoff, and C. Urban. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Proc. of the 7th International Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 69–86, 2016.

**3** J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

**4** T. Coquand and V. Siles. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *Proc. of the 1st International Conference on Certified Programs and Proofs (CPP)*, volume 7086 of *LNCS*, pages 119–134, 2011.

**5** A. Frisch and L. Cardelli. Greedy Regular Expression Matching. In *Proc. of the 31st International Conference on Automata, Languages and Programming (ICALP)*, volume 3142 of *LNCS*, pages 618–629, 2004.

**6** A. Krauss and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *Journal of Automated Reasoning*, 49:95–106, 2012.

**7** C. Kuklewicz. Regex Posix. https://wiki.haskell.org/Regex_Posix.

**8** S. Okui and T. Suzuki. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. In *Proc. of the 15th International Conference on Implementation and Application of Automata (CIAA)*, volume 6482 of *LNCS*, pages 231–240, 2010.

**9** S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.

**10** M. Sulzmann and K. Lu. POSIX Regular Expression Parsing with Derivatives. In *Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS)*, volume 8475 of *LNCS*, pages 203–220, 2014.

**11** S. Vansummeren. Type Inference for Unique Pattern Matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.