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 bitcodes 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 bitcodes is to generate POSIX values incrementally while derivatives are calculated. They also help with designing an "aggressive" simplification function that keeps the size of derivatives small. Without simplification the size derivatives can grow exponentially resulting in an extremely slow lexing algorithm. In this paper we describe a variant of Sulzmann and Lu's algorithm: Our algorithm is a recursive functional program, whereas Sulzmann and Lu's version involves a fixpoint construction. We *(i)* prove in Isabelle/HOL that our program is correct and generates unique POSIX values; we also *(ii)* establish a polynomial bound for the size of the derivatives. The size can be seen as a proxy measure for the efficiency of the lexing algorithm: because of the polynomial bound our algorithm does not suffer from the exponential blowup in earlier works.

2012 ACM Subject Classification Design and analysis of algorithms; Formal languages and automata theory

Keywords and phrases POSIX matching, Derivatives of Regular Expressions, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

In the last fifteen or so years, Brzozowski's derivatives of regular expressions have sparked quite a bit of interest in the functional programming and theorem prover communities. The beauty of Brzozowski's derivatives [3] 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].

The notion of derivatives [3], written $r\backslash c$, of a regular expression give 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 $s \in L\ (r\backslash c)$.

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.

$$\begin{tikzcd}
r_1 \arrow[r, "\_\backslash a"] & r_2 \arrow[r, "\_\backslash b"] & r_3 \arrow[r, "\_\backslash c"] & r_4 \; \mathit{nullable} \arrow[d, "\mathit{mkeps}"] \\
v_1 & v_2 \arrow[l, "\mathit{inj}\ r_1\ a"] & v_3 \arrow[l, "\mathit{inj}\ r_2\ b"] & v_4 \arrow[l, "\mathit{inj}\ r_3\ c"]
\end{tikzcd}$$

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.

$$\frac{}{([], \mathbf{1}) \to \mathit{Empty}}\,P\mathbf{1} \qquad \frac{}{([c], c) \to \mathit{Char}\ c}\,Pc$$

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

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

$$\frac{}{([], r^\star) \to \mathit{Stars}\ []}\,P[]$$

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

Figure 2 Our inductive definition of POSIX values.

$$\begin{array}{rcl}
\mathbf{0}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathbf{1}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
d\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \mathbf{1}\ \mathit{else}\ \mathbf{0} \\
(r_1 + r_2)\backslash c & \stackrel{\text{def}}{=} & (r_1\backslash c) + (r_2\backslash c) \\
(r_1 \cdot r_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}\ r_1 \\
 & & \mathit{then}\ (r_1\backslash c) \cdot r_2 + (r_2\backslash c) \\
 & & \mathit{else}\ (r_1\backslash c) \cdot r_2 \\
(r^\star)\backslash c & \stackrel{\text{def}}{=} & (r\backslash c) \cdot r^\star
\end{array}
\qquad
\begin{array}{lcl}
\mathit{nullable}\ (\mathbf{0}) & \stackrel{\text{def}}{=} & \mathit{False} \\
\mathit{nullable}\ (\mathbf{1}) & \stackrel{\text{def}}{=} & \mathit{True} \\
\mathit{nullable}\ (c) & \stackrel{\text{def}}{=} & \mathit{False} \\
\mathit{nullable}\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}\ r_1 \vee \mathit{nullable}\ r_2 \\
\mathit{nullable}\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}\ r_1 \wedge \mathit{nullable}\ r_2 \\
\mathit{nullable}\ (r^\star) & \stackrel{\text{def}}{=} & \mathit{True}
\end{array}$$

2 Background

Sulzmann-Lu algorithm with inj. State that POSIX rules. metion slg is correct.

$$\begin{array}{lcl}
\mathit{mkeps}\ \mathbf{1} & \stackrel{\text{def}}{=} & \mathit{Empty} \\
\mathit{mkeps}\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{mkeps}\ r_2) \\
\mathit{mkeps}\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}\ r_1\ \mathit{then}\ \mathit{Left}\ (\mathit{mkeps}\ r_1)\ \mathit{else}\ \mathit{Right}\ (\mathit{mkeps}\ r_2) \\
\mathit{mkeps}\ (r^\star) & \stackrel{\text{def}}{=} & \mathit{Stars}\ []
\end{array}$$

$$
\begin{array}{lll}
(1) & \mathit{inj}\ d\ c\ (\mathit{Empty}) & \stackrel{\text{def}}{=} \mathit{Char}\ d \\
(2) & \mathit{inj}\ (r_1 + r_2)\ c\ (\mathit{Left}\ v_1) & \stackrel{\text{def}}{=} \mathit{Left}\ (\mathit{inj}\ r_1\ c\ v_1) \\
(3) & \mathit{inj}\ (r_1 + r_2)\ c\ (\mathit{Right}\ v_2) & \stackrel{\text{def}}{=} \mathit{Right}\ (\mathit{inj}\ r_2\ c\ v_2) \\
(4) & \mathit{inj}\ (r_1 \cdot r_2)\ c\ (\mathit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} \mathit{Seq}\ (\mathit{inj}\ r_1\ c\ v_1)\ v_2 \\
(5) & \mathit{inj}\ (r_1 \cdot r_2)\ c\ (\mathit{Left}\ (\mathit{Seq}\ v_1\ v_2)) & \stackrel{\text{def}}{=} \mathit{Seq}\ (\mathit{inj}\ r_1\ c\ v_1)\ v_2 \\
(6) & \mathit{inj}\ (r_1 \cdot r_2)\ c\ (\mathit{Right}\ v_2) & \stackrel{\text{def}}{=} \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{inj}\ r_2\ c\ v_2) \\
(7) & \mathit{inj}\ (r^\star)\ c\ (\mathit{Seq}\ v\ (\mathit{Stars}\ vs)) & \stackrel{\text{def}}{=} \mathit{Stars}\ (\mathit{inj}\ r\ c\ v :: vs)
\end{array}
$$

3 Bitcoded Regular Expressions and Derivatives

Sulzmann and Lu describe another algorithm that generates POSIX values but dispences with the second phase where characters are injected “back” into values. For this they annotate bitcodes to regular expressions, which we define in Isabelle/HOL as the datatype

$$
\begin{array}{rcl}
\mathit{breg} & ::= & \mathit{ZERO} \\
 & \mid & \mathit{ONE}\ bs \\
 & \mid & \mathit{CHAR}\ bs\ c \\
 & \mid & \mathit{ALTs}\ bs\ rs \\
 & \mid & \mathit{SEQ}\ bs\ r_1\ r_2 \\
 & \mid & \mathit{STAR}\ bs\ r
\end{array}
$$

where *bs* stands for a bitsequences; r, r_1 and r_2 for annotated regular expressions; and *rs* for a list of annotated regular expressions. In contrast to Sulzmann and Lu we generalise the alternative regular expressions to lists, instead of just having binary regular expressions. The idea with annotated regular expressions is to incrementally generate the value information by recording bitsequences. Sulzmann and Lu then define a coding function for how values can be coded into bitsequences.

$$
\begin{array}{lcl}
\mathit{code}\ (\mathit{Empty}) & \stackrel{\text{def}}{=} & [] \\
\mathit{code}\ (\mathit{Char}\ c) & \stackrel{\text{def}}{=} & [] \\
\mathit{code}\ (\mathit{Left}\ v) & \stackrel{\text{def}}{=} & Z :: \mathit{code}\ v \\
\mathit{code}\ (\mathit{Right}\ v) & \stackrel{\text{def}}{=} & S :: \mathit{code}\ v \\
\mathit{code}\ (\mathit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} & \mathit{code}\ v_1\ @\ \mathit{code}\ v_2 \\
\mathit{code}\ (\mathit{Stars}\ []) & \stackrel{\text{def}}{=} & [S] \\
\mathit{code}\ (\mathit{Stars}\ (v :: vs)) & \stackrel{\text{def}}{=} & Z :: \mathit{code}\ v\ @\ \mathit{code}\ (\mathit{Stars}\ vs)
\end{array}
$$

There is also a corresponding decoding function that takes a bitsequence and generates back a value. However, since the bitsequences are a “lossy” coding (*Seq*s are not coded) the decoding function depends also on a regular expression in order to decode values.

$$\begin{array}{lcl}
decode'\ bs\ (\mathbf{1}) & \stackrel{\text{def}}{=} & (Empty,\ bs)\\
decode'\ bs\ (d) & \stackrel{\text{def}}{=} & (Char\ d,\ bs)\\
decode'\ []\ (r1.0 + r2.0) & \stackrel{\text{def}}{=} & (Empty,\ [])\\
decode'\ (Z :: bs)\ (r1.0 + r2.0) & \stackrel{\text{def}}{=} & \text{let}\ (v, y) = decode'\ bs\ r1.0\ \text{in}\ (Left\ v,\ y)\\
decode'\ (S :: bs)\ (r1.0 + r2.0) & \stackrel{\text{def}}{=} & \text{let}\ (v, y) = decode'\ bs\ r2.0\ \text{in}\ (Right\ v,\ y)\\
decode'\ bs\ (r1.0 \cdot r2.0) & \stackrel{\text{def}}{=} & \text{let}\ (v1, bs') = decode'\ bs\ r1.0;\ (v2, y) = decode'\ bs'\ r2.0\ \text{in}\ (Seq\ v1\ v2,\ y)\\
decode'\ []\ (r^\star) & \stackrel{\text{def}}{=} & (Empty,\ [])\\
decode'\ (S :: bs)\ (r^\star) & \stackrel{\text{def}}{=} & (Stars\ [],\ bs)\\
decode'\ (Z :: bs)\ (r^\star) & \stackrel{\text{def}}{=} & \text{let}\ (v, bs') = decode'\ bs\ r;\ (vs, y) = decode'\ bs'\ (r^\star)\ \text{in}\ (Stars\_add\ v\ vs,\ y) \quad \text{fix}
\end{array}$$

The idea of the bitcodes is to annotate them to regular expressions and generate values incrementally. The bitcodes can be read off from the *breg* and then decoded into a value.

$$\begin{array}{lcl}
breg & ::= & ZERO\\
 & \mid & ONE\ bs\\
 & \mid & CHAR\ bs\ c\\
 & \mid & ALTs\ bs\ rs\\
 & \mid & SEQ\ bs\ r_1\ r_2\\
 & \mid & STAR\ bs\ r
\end{array}$$

$$\begin{array}{lcl}
retrieve\ (ONE\ bs)\ (Empty) & \stackrel{\text{def}}{=} & bs\\
retrieve\ (CHAR\ bs\ c)\ (Char\ d) & \stackrel{\text{def}}{=} & bs\\
retrieve\ (ALTs\ bs\ [r])\ v & \stackrel{\text{def}}{=} & bs\ @\ retrieve\ r\ v\\
retrieve\ (ALTs\ bs\ (r :: rs))\ (Left\ v) & \stackrel{\text{def}}{=} & bs\ @\ retrieve\ r\ v\\
retrieve\ (ALTs\ bs\ (r :: rs))\ (Right\ v) & \stackrel{\text{def}}{=} & bs\ @\ retrieve\ (ALTs\ []\ rs)\ v\\
retrieve\ (SEQ\ bs\ r_1\ r_2)\ (Seq\ v_1\ v_2) & \stackrel{\text{def}}{=} & bs\ @\ retrieve\ r_1\ v_1\ @\ retrieve\ r_2\ v_2\\
retrieve\ (STAR\ bs\ r)\ (Stars\ []) & \stackrel{\text{def}}{=} & bs\ @\ [S]\\
retrieve\ (STAR\ bs\ r)\ (Stars\ (v :: vs)) & \stackrel{\text{def}}{=} & bs\ @\ [Z]\ @\ retrieve\ r\ v\ @\ retrieve\ (STAR\ []\ r)\ (Stars\ vs)
\end{array}$$

▶ **Theorem 1.** *blexer r s = lexer r s*

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

4 Simplification

Sulzmann & Lu apply simplification via a fixpoint operation; also does not use erase to filter out duplicates.

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

5 Bound - NO

6 Bounded Regex / Not

7 Conclusion

[2]

$$\frac{}{(\mathit{SEQ}\ bs\ \mathit{ZERO}\ r_2) \rightsquigarrow (\mathit{ZERO})} \qquad \frac{}{(\mathit{SEQ}\ bs\ r_1\ \mathit{ZERO}) \rightsquigarrow (\mathit{ZERO})} \qquad \frac{}{(\mathit{SEQ}\ bs_1\ (\mathit{ONE}\ bs_2)\ r) \rightsquigarrow \mathit{fuse}\ (bs_1\ @\ bs_2)\ r}$$

$$\frac{r_1 \rightsquigarrow r_2}{(\mathit{SEQ}\ bs\ r_1\ r_3) \rightsquigarrow (\mathit{SEQ}\ bs\ r_2\ r_3)} \qquad \frac{r_3 \rightsquigarrow r_4}{(\mathit{SEQ}\ bs\ r_1\ r_3) \rightsquigarrow (\mathit{SEQ}\ bs\ r_1\ r_4)}$$

$$\frac{}{(\mathit{ALTs}\ bs\ []) \rightsquigarrow (\mathit{ZERO})} \qquad \frac{}{(\mathit{ALTs}\ bs\ [r]) \rightsquigarrow \mathit{fuse}\ bs\ r}$$

$$\frac{rs_1 \overset{s}{\rightsquigarrow} rs_2}{(\mathit{ALTs}\ bs\ rs_1) \rightsquigarrow (\mathit{ALTs}\ bs\ rs_2)}$$

$$\frac{}{[] \overset{s}{\rightsquigarrow} []} \qquad \frac{rs_1 \overset{s}{\rightsquigarrow} rs_2}{r :: rs_1 \overset{s}{\rightsquigarrow} r :: rs_2} \qquad \frac{r_1 \rightsquigarrow r_2}{r_1 :: rs \overset{s}{\rightsquigarrow} r_2 :: rs}$$

$$\frac{}{\mathit{ZERO} :: rs \overset{s}{\rightsquigarrow} rs} \qquad \frac{}{\mathit{ALTs}\ bs\ rs_1 :: rs_2 \overset{s}{\rightsquigarrow} (\mathit{map}\ (\mathit{fuse}\ bs)\ rs_1\ @\ rs_2)}$$

$$\frac{r_1{}^{\downarrow} = r_2{}^{\downarrow}}{(rs_1\ @\ [r_1]\ @\ rs_2\ @\ [r_2]\ @\ rs_3) \overset{s}{\rightsquigarrow} (rs_1\ @\ [r_1]\ @\ rs_2\ @\ rs_3)}$$

Figure 3 ???

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.