

# 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 Brzowski 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 finite. Without simplification the size derivatives can grow arbitrarily big 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 finite bound for the size of the derivatives.

**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, Brzowski’s derivatives of regular expressions have sparked quite a bit of interest in the functional programming and theorem prover communities. The beauty of Brzowski’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 Brzowski’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  $r \setminus c$ , 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  $s \in L (r \setminus c)$ .

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  $(x + y + xy)^*$ . 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.



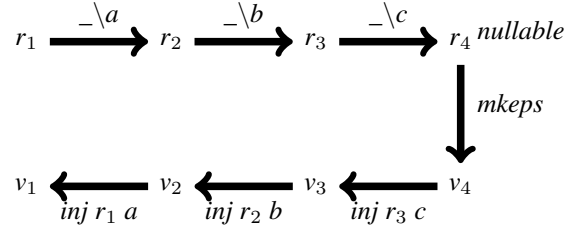
© :

licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **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.

$$\begin{array}{c}
 \frac{}{([\ ], \mathbf{1}) \rightarrow \text{Empty}} P\mathbf{1} \qquad \frac{}{([c], c) \rightarrow \text{Char } c} Pc \\
 \\
 \frac{(s, r_1) \rightarrow v}{(s, r_1 + r_2) \rightarrow \text{Left } v} P+L \qquad \frac{(s, r_2) \rightarrow v \quad s \notin L r_1}{(s, r_1 + r_2) \rightarrow \text{Right } v} P+R \\
 \\
 \frac{\begin{array}{l} (s_1, r_1) \rightarrow v_1 \quad (s_2, r_2) \rightarrow v_2 \\ \nexists s_3 s_4. s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L r_1 \wedge s_4 \in L r_2 \end{array}}{(s_1 @ s_2, r_1 \cdot r_2) \rightarrow \text{Seq } v_1 v_2} PS \\
 \\
 \frac{}{([\ ], r^*) \rightarrow \text{Stars } []} P[] \\
 \\
 \frac{\begin{array}{l} (s_1, r) \rightarrow v \quad (s_2, r^*) \rightarrow \text{Stars } vs \quad |v| \neq [] \\ \nexists s_3 s_4. s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L r \wedge s_4 \in L (r^*) \end{array}}{(s_1 @ s_2, r^*) \rightarrow \text{Stars } (v :: vs)} P\star
 \end{array}$$

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

$$\begin{array}{ll}
 \mathbf{0} \setminus c \stackrel{\text{def}}{=} \mathbf{0} & \text{nullable } (\mathbf{0}) \stackrel{\text{def}}{=} \text{False} \\
 \mathbf{1} \setminus c \stackrel{\text{def}}{=} \mathbf{0} & \text{nullable } (\mathbf{1}) \stackrel{\text{def}}{=} \text{True} \\
 d \setminus c \stackrel{\text{def}}{=} \text{if } c = d \text{ then } \mathbf{1} \text{ else } \mathbf{0} & \text{nullable } (c) \stackrel{\text{def}}{=} \text{False} \\
 (r_1 + r_2) \setminus c \stackrel{\text{def}}{=} (r_1 \setminus c) + (r_2 \setminus c) & \text{nullable } (r_1 + r_2) \stackrel{\text{def}}{=} \text{nullable } r_1 \vee \text{nullable } r_2 \\
 (r_1 \cdot r_2) \setminus c \stackrel{\text{def}}{=} \text{if nullable } r_1 & \text{nullable } (r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{nullable } r_1 \wedge \text{nullable } r_2 \\
 \quad \text{then } (r_1 \setminus c) \cdot r_2 + (r_2 \setminus c) & \text{nullable } (r^*) \stackrel{\text{def}}{=} \text{True} \\
 \quad \text{else } (r_1 \setminus c) \cdot r_2 & \\
 (r^*) \setminus c \stackrel{\text{def}}{=} (r \setminus c) \cdot r^* &
 \end{array}$$

## 2 Background

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

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

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

### 3 Bitcoded Regular Expressions and Derivatives

In the second part of their paper [10], 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

$$\begin{aligned} \text{breg} \quad ::= & \text{ZERO} \quad | \quad \text{ONE } bs \\ & | \quad \text{CHAR } bs \ c \\ & | \quad \text{ALTS } bs \ rs \\ & | \quad \text{SEQ } bs \ r_1 \ r_2 \\ & | \quad \text{STAR } bs \ r \end{aligned}$$

where  $bs$  stands for bitsequences;  $r$ ,  $r_1$  and  $r_2$  for bitcoded regular expressions; and  $rs$  for lists of bitcoded regular expressions. The binary alternative  $\text{ALT } bs \ r_1 \ r_2$  is just an abbreviation for  $\text{ALTS } bs \ [r_1, r_2]$ . For bitsequences we just use lists made up of the constants  $Z$  and  $S$ . The idea with bitcoded regular expressions is to incrementally generate the value information (for example  $\text{Left}$  and  $\text{Right}$ ) as bitsequences. For this Sulzmann and Lu define a coding function for how values can be coded into bitsequences.

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

As can be seen, this coding is “lossy” in the sense that we do not record explicitly character values and also not sequence values (for them we just append two bitsequences). However, the different alternatives for  $\text{Left}$ , respectively  $\text{Right}$ , are recorded as  $Z$  and  $S$  followed by some bitsequence. Similarly, we use  $Z$  to indicate if there is still a value coming in the list of  $\text{Stars}$ , whereas  $S$  indicates the end of the list. The lossiness makes the process of decoding a bit more involved, but the point is that if we have a regular expression *and* a bitsequence of a corresponding value, then we can always decode the value accurately. The decoding can be defined by using two functions called  $\text{decode}'$  and  $\text{decode}$ :

## XX:4 POSIX Lexing with Bitcoded Derivatives

$$\begin{aligned}
\text{decode}' \text{ bs } (\mathbf{1}) & \stackrel{\text{def}}{=} (\text{Empty}, \text{bs}) \\
\text{decode}' \text{ bs } (c) & \stackrel{\text{def}}{=} (\text{Char } c, \text{bs}) \\
\text{decode}' (Z :: \text{bs}) (r_1 + r_2) & \stackrel{\text{def}}{=} \text{let } (v, \text{bs}_1) = \text{decode}' \text{ bs } r_1 \text{ in } (\text{Left } v, \text{bs}_1) \\
\text{decode}' (S :: \text{bs}) (r_1 + r_2) & \stackrel{\text{def}}{=} \text{let } (v, \text{bs}_1) = \text{decode}' \text{ bs } r_2 \text{ in } (\text{Right } v, \text{bs}_1) \\
\text{decode}' \text{ bs } (r_1 \cdot r_2) & \stackrel{\text{def}}{=} \text{let } (v_1, \text{bs}_1) = \text{decode}' \text{ bs } r_1 \text{ in} \\
& \quad \text{let } (v_2, \text{bs}_2) = \text{decode}' \text{ bs}_1 r_2 \text{ in } (\text{Seq } v_1 v_2, \text{bs}_2) \\
\text{decode}' (Z :: \text{bs}) (r^*) & \stackrel{\text{def}}{=} (\text{Stars } [], \text{bs}) \\
\text{decode}' (S :: \text{bs}) (r^*) & \stackrel{\text{def}}{=} \text{let } (v, \text{bs}_1) = \text{decode}' \text{ bs } r \text{ in} \\
& \quad \text{let } (\text{Stars } \text{vs}, \text{bs}_2) = \text{decode}' \text{ bs}_1 r^* \text{ in } (\text{Stars } v :: \text{vs}, \text{bs}_2) \\
\text{decode } \text{ bs } r & \stackrel{\text{def}}{=} \text{let } (v, \text{bs}') = \text{decode}' \text{ bs } r \text{ in} \\
& \quad \text{if } \text{bs}' = [] \text{ then } \text{Some } v \text{ else } \text{None}
\end{aligned}$$

The function *decode* checks whether all of the bitsequence is consumed and returns the corresponding value as *Some v*; otherwise it fails with *None*. We can establish that for a value *v* inhabited by a regular expression *r*, the decoding of its bitsequence never fails.

► **Lemma 1.** *If  $\vdash v : r$  then  $\text{decode}(\text{code } v) r = \text{Some } v$ .*

**Proof.** This follows from the property that  $\text{decode}'((\text{code } v) @ \text{bs}) r = (v, \text{bs})$  holds for any bitsequence *bs* and  $\vdash v : r$ . This property can be easily proved by induction on  $\vdash v : r$ . ◀

Sulzmann and Lu define the function *internalise* in order to transform standard regular expressions into annotated regular expressions. We write this operation as  $r^\uparrow$ . This internalisation uses the following *fuse* function.

$$\begin{aligned}
\text{fuse } \text{bs } (\text{ZERO}) & \stackrel{\text{def}}{=} \text{ZERO} \\
\text{fuse } \text{bs } (\text{ONE } \text{bs}') & \stackrel{\text{def}}{=} \text{ONE } (\text{bs } @ \text{bs}') \\
\text{fuse } \text{bs } (\text{CHAR } \text{bs}' c) & \stackrel{\text{def}}{=} \text{CHAR } (\text{bs } @ \text{bs}') c \\
\text{fuse } \text{bs } (\text{ALTS } \text{bs}' r s) & \stackrel{\text{def}}{=} \text{ALTS } (\text{bs } @ \text{bs}') r s \\
\text{fuse } \text{bs } (\text{SEQ } \text{bs}' r_1 r_2) & \stackrel{\text{def}}{=} \text{SEQ } (\text{bs } @ \text{bs}') r_1 r_2 \\
\text{fuse } \text{bs } (\text{STAR } \text{bs}' r) & \stackrel{\text{def}}{=} \text{STAR } (\text{bs } @ \text{bs}') r
\end{aligned}$$

A regular expression can then be *internalised* into a bitcoded regular expression as follows.

$$\begin{aligned}
(\mathbf{0})^\uparrow & \stackrel{\text{def}}{=} \text{ZERO} \\
(\mathbf{1})^\uparrow & \stackrel{\text{def}}{=} \text{ONE } [] \\
(c)^\uparrow & \stackrel{\text{def}}{=} \text{CHAR } [] c \\
(r_1 + r_2)^\uparrow & \stackrel{\text{def}}{=} \text{ALT } [] (\text{fuse } [Z] r_1^\uparrow) (\text{fuse } [S] r_2^\uparrow) \\
(r_1 \cdot r_2)^\uparrow & \stackrel{\text{def}}{=} \text{SEQ } [] r_1^\uparrow r_2^\uparrow \\
(r^*)^\uparrow & \stackrel{\text{def}}{=} \text{STAR } [] r^\uparrow
\end{aligned}$$

There is also an *erase*-function, written  $a^\downarrow$ , which transforms a bitcoded regular expression into a (standard) regular expression by just erasing the annotated bitsequences. We omit the straightforward definition. For defining the algorithm, we also need the functions *bnullable* and *bmkeps*, which are the “lifted” versions of *nullable* and *mkeps* acting on bitcoded regular expressions, instead of regular expressions.

$bnullable (ZERO)$	$\stackrel{\text{def}}{=} false$	$bmkeps (ONE bs)$	$\stackrel{\text{def}}{=} bs$
$bnullable (ONE bs)$	$\stackrel{\text{def}}{=} true$	$bmkeps (ALTs bs r :: rs)$	$\stackrel{\text{def}}{=} \text{if } bnullable r$
$bnullable (CHAR bs c)$	$\stackrel{\text{def}}{=} false$		$\text{then } bs @ bmkeps r$
$bnullable (ALTs bs rs)$	$\stackrel{\text{def}}{=} \exists r \in rs. bnullable r$	$bmkeps (SEQ bs r_1 r_2)$	$\stackrel{\text{def}}{=} \text{else } bs @ bmkeps rs$
$bnullable (SEQ bs r_1 r_2)$	$\stackrel{\text{def}}{=} bnullable r_1 \wedge bnullable r_2$		$bs @ bmkeps r_1 @ bmkeps r_2$
$bnullable (STAR bs r)$	$\stackrel{\text{def}}{=} true$	$bmkeps (STAR bs r)$	$\stackrel{\text{def}}{=} bs @ [S]$

The key function in the bitcoded algorithm is the derivative of an bitcoded regular expression. This derivative calculates the derivative but at the same time also the incremental part of bitsequences that contribute to constructing a POSIX value.

$(ZERO) \setminus c$	$\stackrel{\text{def}}{=} ZERO$
$(ONE bs) \setminus c$	$\stackrel{\text{def}}{=} ZERO$
$(CHAR bs d) \setminus c$	$\stackrel{\text{def}}{=} \text{if } c = d \text{ then } ONE bs \text{ else } ZERO$
$(ALTs bs rs) \setminus c$	$\stackrel{\text{def}}{=} ALTs bs (map (\_ \setminus c) rs)$
$(SEQ bs r_1 r_2) \setminus c$	$\stackrel{\text{def}}{=} \text{if } bnullable r_1$
	$\text{then } ALT bs (SEQ [] (r_1 \setminus c) r_2)$
	$(fuse (bmkeps r_1) (r_2 \setminus c))$
	$\text{else } SEQ bs (r_1 \setminus c) r_2$
$(STAR bs r) \setminus c$	$\stackrel{\text{def}}{=} SEQ bs (fuse [Z](r \setminus c)) (STAR [] r)$

This function can also be extended to strings, written  $r \setminus s$ , just like the standard derivative. We omit the details. Finally we can define Sulzmann and Lu's bitcoded lexer, which we call *blexer*:

$blexer r s$	$\stackrel{\text{def}}{=} \text{let } r_{der} = (r^\dagger) \setminus s \text{ in}$
	$\text{if } bnullable(r_{der}) \text{ then } decode (bmkeps r_{der}) r \text{ else } None$

This bitcoded lexer first internalises the regular expression  $r$  and then builds the bitcoded derivative according to  $s$ . If the derivative is (b)nullable the string is in the language of  $r$  and it extracts the bitsequence using the *bmkeps* function. Finally it decodes the bitsequence into a value. If the derivative is *not* nullable, then *None* is returned. We can show that this way of calculating a value generates the same result as with *lexer*.

Before we can proceed we need to define a helper function, called *retrieve*, which Sulzmann and Lu introduced for the correctness proof.

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

The idea behind this function is to retrieve a possibly partial bitcode from a bitcoded regular expression, where the retrieval is guided by a value. For example if the value is *Left* then we descend into the left-hand side of an alternative in order to assemble the bitcode. Similarly for *Right*. The property

## XX:6 POSIX Lexing with Bitcoded Derivatives

we can show is that for a given  $v$  and  $r$  with  $\vdash v : r$ , the retrieved bitsequence from the internalised regular expression is equal to the bitcoded version of  $v$ .

► **Lemma 2.** *If  $\vdash v : r$  then  $\text{code } v = \text{retrieve } (r^\dagger) v$ .*

We also need some auxiliary facts about how the bitcoded operations relate to the “standard” operations on regular expressions. For example if we build a bitcoded derivative and erase the result, this is the same as if we first erase the bitcoded regular expression and then perform the “standard” derivative operation.

► **Lemma 3.**

- (1)  $(a \setminus s)^\dagger = (a^\dagger) \setminus s$
- (2)  $\text{bnutable}(a)$  iff  $\text{nutable}(a^\dagger)$
- (3)  $\text{bmkeps}(a) = \text{retrieve } a (\text{mkeps } (a^\dagger))$  provided  $\text{nutable}(a^\dagger)$ .

**Proof.** All properties are by induction on annotated regular expressions. There are no interesting cases. ◀

This brings us to our main lemma in this section: if we build a derivative, say  $r \setminus s$  and have a value, say  $v$ , inhabited by this derivative, then we can produce the result *lexer* generates by applying this value to the stacked-up injection functions *flex* assembles. The lemma establishes that this is the same value as if we build the annotated derivative  $r^\dagger \setminus s$  and then retrieve the corresponding bitcoded version, followed by a decoding step.

► **Lemma 4 (Main Lemma).** *If  $\vdash v : r \setminus s$  then*

$$\text{Some } (\text{flex } r \text{ id } s \ v) = \text{decode}(\text{retrieve } (r^\dagger \setminus s) \ v) \ r$$

**Proof.** This can be proved by induction on  $s$  and generalising over  $v$ . The interesting point is that we need to prove this in the reverse direction for  $s$ . This means instead of cases  $[]$  and  $c :: s$ , we have cases  $[]$  and  $s @ [c]$  where we unravel the string from the back.<sup>1</sup>

The case for  $[]$  is routine using Lemmas 1 and 2. In the case  $s @ [c]$ , we can infer from the assumption that  $\vdash v : (r \setminus s) \setminus c$  holds. Hence by Lemma ?? we know that  $(*) \vdash \text{inj } (r \setminus s) \ c \ v : r \setminus s$  holds too. By definition of *flex* we can unfold the left-hand side to be

$$\text{Some } (\text{flex } r \text{ id } (s @ [c]) \ v) = \text{Some } (\text{flex } r \text{ id } s \ (\text{inj } (r \setminus s) \ c \ v))$$

By induction hypothesis and  $(*)$  we can rewrite the right-hand side to

$$\text{decode } (\text{retrieve } (r^\dagger \setminus s) \ (\text{inj } (r \setminus s) \ c \ v)) \ r$$

which is equal to  $\text{decode } (\text{retrieve } (r^\dagger \setminus (s @ [c])) \ v) \ r$  as required. The last rewrite step is possible because we generalised over  $v$  in our induction. ◀

With this lemma in place, we can prove the correctness of *blexer* such that it produces the same result as *lexer*.

► **Theorem 5.**  *$\text{lexer } r \ s = \text{blexer } r \ s$*

---

<sup>1</sup> Isabelle/HOL provides an induction principle for this way of performing the induction.

**Proof.** We can first expand both sides using Lemma ?? and the definition of *blexer*. This gives us two *if*-statements, which we need to show to be equal. By Lemma 3(2) we know the *if*-tests coincide:

$$bnullable(r^\uparrow \setminus s) \text{ iff } nullable(r \setminus s)$$

For the *if*-branch suppose  $r_d \stackrel{\text{def}}{=} r^\uparrow \setminus s$  and  $d \stackrel{\text{def}}{=} r \setminus s$ . We have (\*) *nullable*  $d$ . We can then show by Lemma 3(3) that

$$\text{decode}(bmkeps\ r_d)\ r = \text{decode}(\text{retrieve}\ a\ (mkeps\ d))\ r$$

where the right-hand side is equal to *Some (flex rid s (mkeps d))* by Lemma 4 (we know  $\vdash mkeps\ d : d$  by (\*)). This shows the *if*-branches return the same value. In the *else*-branches both *lexer* and *blexer* return *None*. Therefore we can conclude the proof. ◀

This establishes that the bitcoded algorithm by Sulzmann and Lu without simplification produces correct results. This was only conjectured in their paper [10]. The next step is to add simplifications.

## 4 Simplification

- ▶ **Lemma 6.** *If*  $r_1 \rightsquigarrow r_2$  *then*  $bnullable\ r_1 = bnullable\ r_2$ .
- ▶ **Lemma 7.** *If*  $r_1 \rightsquigarrow r_2$  *and*  $bnullable\ r_1$  *then*  $bmkeps\ r_1 = bmkeps\ r_2$ .
- ▶ **Lemma 8.**  $r \rightsquigarrow^* bsimp\ r$
- ▶ **Lemma 9.** *If*  $r_1 \rightsquigarrow r_2$  *then*  $r_1 \setminus c \rightsquigarrow^* r_2 \setminus c$ .
- ▶ **Lemma 10.**  $bders\ r\ s \rightsquigarrow^* bders\_simp\ r\ s$
- ▶ **Theorem 11.**  $blexer\ r\ s = blexer^+\ r\ s$

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]

---

### 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](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.

$$\begin{array}{c}
 \frac{}{(SEQ\ bs\ ZERO\ r_2) \rightsquigarrow (ZERO)} \quad \frac{}{(SEQ\ bs\ r_1\ ZERO) \rightsquigarrow (ZERO)} \quad \frac{}{(SEQ\ bs_1\ (ONE\ bs_2)\ r) \rightsquigarrow fuse\ (bs_1\ @\ bs_2)\ r} \\
 \frac{r_1 \rightsquigarrow r_2}{(SEQ\ bs\ r_1\ r_3) \rightsquigarrow (SEQ\ bs\ r_2\ r_3)} \quad \frac{r_3 \rightsquigarrow r_4}{(SEQ\ bs\ r_1\ r_3) \rightsquigarrow (SEQ\ bs\ r_1\ r_4)} \\
 \frac{}{(ALts\ bs\ []) \rightsquigarrow (ZERO)} \quad \frac{}{(ALts\ bs\ [r]) \rightsquigarrow fuse\ bs\ r} \\
 \frac{rs_1 \rightsquigarrow^s rs_2}{(ALts\ bs\ rs_1) \rightsquigarrow (ALts\ bs\ rs_2)} \\
 \frac{rs_1 \rightsquigarrow^s rs_2}{r :: rs_1 \rightsquigarrow^s r :: rs_2} \quad \frac{r_1 \rightsquigarrow r_2}{r_1 :: rs \rightsquigarrow^s r_2 :: rs} \\
 \frac{}{ZERO :: rs \rightsquigarrow^s rs} \quad \frac{}{ALts\ bs\ rs_1 :: rs_2 \rightsquigarrow^s (map\ (fuse\ bs)\ rs_1\ @\ rs_2)} \\
 \frac{L(r_1^\downarrow) \subseteq L(r_2^\downarrow)}{(rs_1\ @\ [r_2])\ @\ rs_2\ @\ [r_1]\ @\ rs_3 \rightsquigarrow^s (rs_1\ @\ [r_2])\ @\ rs_2\ @\ rs_3)}
 \end{array}$$

■ **Figure 3** ???

- 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](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.