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.

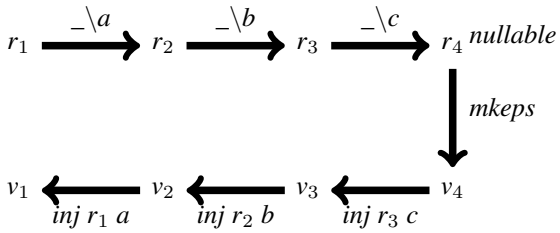

Figure 1 The two phases of the algorithm by Sulzmann & Lu [10], matching the string $[a, b, c]$. The first phase (the arrows from left to right) is Brzozowski's matcher building successive derivatives. If the last regular expression is *nullable*, then the functions of the second phase are called (the 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.

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

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

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

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

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

Figure 2 Our inductive definition of POSIX values.

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

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

2 Background

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

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

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

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

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

where bs stands for a bitsequences; r, r_1 and r_2 for bitcoded regular expressions; and rs for lists of bitcoded regular expressions. The binary alternative $ALT\ bs\ r_1\ r_2$ is just an abbreviation for $ALTs\ bs\ [r_1, r_2]$. 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 *Left* and *Right*) as bitsequences as part of the regular expression constructors. Sulzmann and Lu then define a coding function for how values can be coded into bitsequences.

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

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). We do, however, record the different alternatives for *Left*, respectively *Right*, 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 *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 $decode'$ and *decode*:

$$
\begin{array}{lcl}
\mathit{decode}'\ bs\ (\mathbf{1}) & \stackrel{\text{def}}{=} & (\mathit{Empty}, bs) \\
\mathit{decode}'\ bs\ (c) & \stackrel{\text{def}}{=} & (\mathit{Char}\ c, bs) \\
\mathit{decode}'\ (Z :: bs)\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{let}\ (v, bs_1) = \mathit{decode}'\ bs\ r_1\ \mathit{in}\ (\mathit{Left}\ v, bs_1) \\
\mathit{decode}'\ (S :: bs)\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{let}\ (v, bs_1) = \mathit{decode}'\ bs\ r_2\ \mathit{in}\ (\mathit{Right}\ v, bs_1) \\
\mathit{decode}'\ bs\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{let}\ (v_1, bs_1) = \mathit{decode}'\ bs\ r_1\ \mathit{in} \\
 & & \mathit{let}\ (v_2, bs_2) = \mathit{decode}'\ bs_1\ r_2 \quad \mathit{in}\ (\mathit{Seq}\ v_1\ v_2, bs_2) \\
\mathit{decode}'\ (Z :: bs)\ (r^*) & \stackrel{\text{def}}{=} & (\mathit{Stars}\ [], bs) \\
\mathit{decode}'\ (S :: bs)\ (r^*) & \stackrel{\text{def}}{=} & \mathit{let}\ (v, bs_1) = \mathit{decode}'\ bs\ r\ \mathit{in} \\
 & & \mathit{let}\ (\mathit{Stars}\ vs, bs_2) = \mathit{decode}'\ bs_1\ r^* \quad \mathit{in}\ (\mathit{Stars}\ v :: vs, bs_2) \\
 & & \\
\mathit{decode}\ bs\ r & \stackrel{\text{def}}{=} & \mathit{let}\ (v, bs') = \mathit{decode}'\ bs\ r\ \mathit{in} \\
 & & \quad \mathit{if}\ bs' = []\ \mathit{then}\ \mathit{Some}\ v\ \mathit{else}\ \mathit{None}
\end{array}
$$

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* $\mathit{decode}\,(\mathit{code}\,v)\,r = \mathit{Some}\,v$.

Proof. This follows from the property that $\mathit{decode}'\,((\mathit{code}\,v) \mathbin{@} bs)\,r = (v, bs)$ 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.

$$
\begin{array}{lcl}
\mathit{fuse}\ bs\ (\mathit{ZERO}) & \stackrel{\text{def}}{=} & \mathit{ZERO} \\
\mathit{fuse}\ bs\ (\mathit{ONE}\ bs') & \stackrel{\text{def}}{=} & \mathit{ONE}\ (bs \mathbin{@} bs') \\
\mathit{fuse}\ bs\ (\mathit{CHAR}\ bs'\ c) & \stackrel{\text{def}}{=} & \mathit{CHAR}\ (bs \mathbin{@} bs')\ c \\
\mathit{fuse}\ bs\ (\mathit{ALTs}\ bs'\ rs) & \stackrel{\text{def}}{=} & \mathit{ALTs}\ (bs \mathbin{@} bs')\ rs \\
\mathit{fuse}\ bs\ (\mathit{SEQ}\ bs'\ r_1\ r_2) & \stackrel{\text{def}}{=} & \mathit{SEQ}\ (bs \mathbin{@} bs')\ r_1\ r_2 \\
\mathit{fuse}\ bs\ (\mathit{STAR}\ bs'\ r) & \stackrel{\text{def}}{=} & \mathit{STAR}\ (bs \mathbin{@} bs')\ r
\end{array}
$$

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

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

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.

$$
\begin{array}{lcl}
\mathit{bnullable}\,(\mathit{ZERO}) & \stackrel{\text{def}}{=} & \mathit{false}\ \textbf{fix} \\
\mathit{bnullable}\,(\mathit{ONE}\,bs) & \stackrel{\text{def}}{=} & \mathit{true} \\
\mathit{bnullable}\,(\mathit{CHAR}\,bs\,c) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{bnullable}\,(\mathit{ALT}\,bs\,a_1\,a_2) & \stackrel{\text{def}}{=} & \mathit{bnullable}\,a_1 \vee \mathit{bnullable}\,a_2 \\
\mathit{bnullable}\,(\mathit{SEQ}\,bs\,a_1\,a_2) & \stackrel{\text{def}}{=} & \mathit{bnullable}\,a_1 \wedge \mathit{bnullable}\,a_2 \\
\mathit{bnullable}\,(\mathit{STAR}\,bs\,a) & \stackrel{\text{def}}{=} & \mathit{true}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{bmkeps}\,(\mathit{ONE}\,bs) & \stackrel{\text{def}}{=} & bs\ \textbf{fix} \\
\mathit{bmkeps}\,(\mathit{ALT}\,bs\,a_1\,a_2) & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{bnullable}\,a_1 \\
 & & \mathit{then}\ bs \mathbin{@} \mathit{bmkeps}\,a_1 \\
 & & \mathit{else}\ bs \mathbin{@} \mathit{bmkeps}\,a_2 \\
\mathit{bmkeps}\,(\mathit{SEQ}\,bs\,a_1\,a_2) & \stackrel{\text{def}}{=} & bs \mathbin{@} \mathit{bmkeps}\,a_1 \mathbin{@} \mathit{bmkeps}\,a_2 \\
\mathit{bmkeps}\,(\mathit{STAR}\,bs\,a) & \stackrel{\text{def}}{=} & bs \mathbin{@} [S]
\end{array}
$$

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

$$
\begin{array}{lcl}
(\mathit{ZERO})\backslash c & \stackrel{\text{def}}{=} & \mathit{ZERO}\ \textbf{fix} \\
(\mathit{ONE}\ bs)\backslash c & \stackrel{\text{def}}{=} & \mathit{ZERO} \\
(\mathit{CHAR}\ bs\ d)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \ \mathit{then}\ \mathit{ONE}\ bs\ \mathit{else}\ \mathit{ZERO} \\
(\mathit{ALT}\ bs\ a_1\ a_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{ALT}\ bs\ (a_1\backslash c)\ (a_2\backslash c) \\
(\mathit{SEQ}\ bs\ a_1\ a_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{bnullable}\,a_1 \\
 & & \mathit{then}\ \mathit{ALT}\,bs\ (\mathit{SEQ}\ []\ (a_1\backslash c)\ a_2) \\
 & & \qquad\qquad (\mathit{fuse}\ (\mathit{bmkeps}\,a_1)\ (a_2\backslash c)) \\
 & & \mathit{else}\ \mathit{SEQ}\,bs\ (a_1\backslash c)\ a_2 \\
(\mathit{STAR}\,bs\,a)\backslash c & \stackrel{\text{def}}{=} & \mathit{SEQ}\ bs\ (\mathit{fuse}\ [Z](r\backslash c))\ (\mathit{STAR}\ []\ r)
\end{array}
$$

This function can also be extended to strings, written $a\backslash s$, just like the standard derivative. We omit the details. Finally we can define Sulzmann and Lu's bitcoded lexer, which we call *blexer*:
This bitcoded lexer first internalises the regular expression r and then builds the annotated derivative according to s. If the derivative is nullable, then it extracts the bitcoded value using the *bmkeps* function. Finally it decodes the bitcoded value. If the derivative is *not* nullable, then *None* is returned. The task is to show that this way of calculating a value generates the same result as with *lexer*.

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

fix

The idea behind this function is to retrieve a possibly partial bitcode from an annotated 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 (annotated) regular expression in order to assemble the bitcode. Similarly for $Right$. The property 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* $\mathit{code}\,v = \mathit{retrieve}\,(r^{\uparrow})\,v$.

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.

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

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

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

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

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

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

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

Figure 3 ???

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.

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

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

▶ **Theorem 3.** *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]

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.