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 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 algorithm 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, 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. Derivatives of a regular expression, written $r\backslash c$, 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*). We are aware of a mechanised correctness proof of Brzozowski's matcher in HOL4 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].

There are two difficulties with derivative-based matchers and also lexers: First, Brzozowski's original matcher only generates a yes/no answer for whether a regular expression matches a string or not. Sulzmann and Lu [10] overcome this difficulty by cleverly extending Brzozowski's matching algorithm to POSIX lexing. This extended version generates additional information on *how* a regular expression matches a string. They achieve this by

The second problem is that Brzozowski's derivatives can grow to arbitrarily big sizes. For example if we start with the regular expression $(a + aa)^*$ and take successive derivatives according to the character a, we end up with a sequence of ever-growing derivatives like

$$\begin{array}{lcl}
(a+aa)^* & \xrightarrow{\_\backslash a} & (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^* \\
 & \xrightarrow{\_\backslash a} & (\mathbf{0}+\mathbf{0}a+\mathbf{1})\cdot(a+aa)^* \;+\; (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^* \\
 & \xrightarrow{\_\backslash a} & (\mathbf{0}+\mathbf{0}a+\mathbf{0})\cdot(a+aa)^* + (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^* + \\
 & & \quad (\mathbf{0}+\mathbf{0}a+\mathbf{1})\cdot(a+aa)^* + (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^* \\
 & \xrightarrow{\_\backslash a} & \ldots
\end{array}$$

where after around 35 steps we run out of memory on a typical computer (we define the precise details of the derivative operation later). Clearly, the notation involving $\mathbf{0}$s and $\mathbf{1}$s already suggests simplification rules that can be applied to regular regular expressions, for example $\mathbf{0}r \Rightarrow \mathbf{0}$, $\mathbf{1}r \Rightarrow r$, $\mathbf{0} + r \Rightarrow r$ and $r + r \Rightarrow r$. While such simple-minded reductions have been proved in our earlier work to preserve the correctness of Sulzmann and Lu's algorithm, they unfortunately do *not* help with limiting the gowth of the derivatives shown above: yes, the growth is slowed, but the derivatives can still grow beyond any finite bound.

Sulzmann and Lu introduce a bitcoded version of their lexing algorithm. They make some claims about the correctness and speed of this version, but do not provide any supporting proof arguments, not even "pencil-and-paper" arguments. They wrote about their bitcoded "incremental parsing method" (that is the algorithm to be studied in this section):

> *"Correctness Claim: We further claim that the incremental parsing method [..] in combination with the simplification steps [..] yields POSIX parse trees. We have tested this claim extensively [..] but yet have to work out all proof details."*

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.

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 $s \in L\,(r\backslash c)$.

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

2 *Background*

In our Isabelle/HOL formalisation strings are lists of characters with the empty string being represented by the empty list, written $[]$, and list-cons being written as $\_::\_$; string concatenation is $\_ @ \_$. Often we use the usual bracket notation for lists also for strings; for example a string consisting of just a single character c is written $[c]$. Our egular expressions are defined as usual as the elements of the following inductive datatype:

$$r := \mathbf{0} \mid \mathbf{1} \mid c \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^\star$$

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

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.

where **0** stands for the regular expression that does not match any string, **1** for the regular expression that matches only the empty string and c for matching a character literal. The language of a regular expression, written L, is defined as usual (see for example [2]).

Central to Brzozowski's regular expression matcher are two functions called $\mathit{nullable}$ and *derivative*. The latter is written $r\backslash c$ for the derivative of the regular expression r w.r.t. the character c. Both functions are defined by recursion over regular expressions.

$$\begin{array}{lcl}
\mathit{nullable}(\mathbf{0}) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{nullable}(\mathbf{1}) & \stackrel{\text{def}}{=} & \mathit{true} \\
\mathit{nullable}(c) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{nullable}(r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}(r_1) \vee \mathit{nullable}(r_2) \\
\mathit{nullable}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2) \\
\mathit{nullable}(r^*) & \stackrel{\text{def}}{=} & \mathit{true}
\end{array}$$

The derivative function takes a regular expression, say r and a character, say c, as input and returns the derivative regular expression.

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

Sulzmann and Lu presented two lexing algorithms in their paper from 2014 [10]. This first algorithm consists of two phases: first a matching phase (which is Brzozowski's algorithm) and then a value construction phase. The values encode *how* a regular expression matches a string. *Values* are defined as the inductive datatype

$$v := \mathit{Empty} \mid \mathit{Char}\ c \mid \mathit{Left}\ v \mid \mathit{Right}\ v \mid \mathit{Seq}\ v_1\ v_2 \mid \mathit{Stars}\ \mathit{vs}$$

where we use *vs* to stand for a list of values.
Sulzmann and Lu also define inductively an inhabitation relation that associates values to regular expressions:

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

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

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

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

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

Figure 2 Our inductive definition of POSIX values.

$$\frac{}{\vdash Empty : \mathbf{1}} \qquad \frac{}{\vdash Char\ c : c}$$

$$\frac{\vdash v_1 : r_1}{\vdash Left\ v_1 : r_1 + r_2} \qquad \frac{\vdash v_2 : r_1}{\vdash Right\ v_2 : r_2 + r_1}$$

$$\frac{\vdash v_1 : r_1 \qquad \vdash v_2 : r_2}{\vdash Seq\ v_1\ v_2 : r_1 \cdot r_2}$$

$$\frac{\forall v \in r.\ \vdash v : vs \wedge |v| \neq []}{\vdash Stars\ r : vs^\star}$$

Note that no values are associated with the regular expression $\mathbf{0}$. It is routine to establish how values "inhabiting" a regular expression correspond to the language of a regular expression, namely

▶ **Proposition 1.** $L\ r = \{|v| \mid \vdash v : r\}$

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

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

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

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

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

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 $\mathit{ALT}\ \mathit{bs}\ r_1\ r_2$ is just an abbreviation for $\mathit{ALTs}\ \mathit{bs}\ [r_1, r_2]$. 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. For this Sulzmann and Lu define a coding function for how values can be coded into bitsequences.

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

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). However, the different alternatives for *Left*, respectively *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 *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 2.** *If* $\vdash v : r$ *then* $\mathit{decode}\,(\mathit{code}\,v)\,r = \mathit{Some}\,v$.

Proof. This follows from the property that $\mathit{decode}'\,((\mathit{code}\,v)\,@\,bs)\,r = (v, bs)$ holds for any bit-sequence 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.

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

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.

$$\begin{array}{lcl}
\mathit{bnullable}\,(\mathit{ZERO}) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{bnullable}\,(\mathit{ONE}\,bs) & \stackrel{\text{def}}{=} & \mathit{true} \\
\mathit{bnullable}\,(\mathit{CHAR}\,bs\,c) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{bnullable}\,(\mathit{ALTs}\,bs\,rs) & \stackrel{\text{def}}{=} & \exists r \in rs.\,\mathit{bnullable}\,r \\
\mathit{bnullable}\,(\mathit{SEQ}\,bs\,r_1\,r_2) & \stackrel{\text{def}}{=} & \mathit{bnullable}\,r_1 \wedge \mathit{bnullable}\,r_2 \\
\mathit{bnullable}\,(\mathit{STAR}\,bs\,r) & \stackrel{\text{def}}{=} & \mathit{true}
\end{array}$$

$$\begin{array}{lcl}
\mathit{bmkeps}\,(\mathit{ONE}\,bs) & \stackrel{\text{def}}{=} & bs \\
\mathit{bmkeps}\,(\mathit{ALTs}\,bs\,r :: rs) & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{bnullable}\,r \\
 & & \mathit{then}\ bs\,@\,\mathit{bmkeps}\,r \\
 & & \mathit{else}\ bs\,@\,\mathit{bmkeps}\,rs \\
\mathit{bmkeps}\,(\mathit{SEQ}\,bs\,r_1\,r_2) & \stackrel{\text{def}}{=} & \\
 & & bs\,@\,\mathit{bmkeps}\,r_1\,@\,\mathit{bmkeps}\,r_2 \\
\mathit{bmkeps}\,(\mathit{STAR}\,bs\,r) & \stackrel{\text{def}}{=} & bs\,@\,[S]
\end{array}$$

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.

$$
\begin{array}{lcl}
(ZERO)\backslash c & \stackrel{\text{def}}{=} & ZERO \\
(ONE\ bs)\backslash c & \stackrel{\text{def}}{=} & ZERO \\
(CHAR\ bs\ d)\backslash c & \stackrel{\text{def}}{=} & \textit{if } c = d \ \textit{ then } ONE\ bs \textit{ else } ZERO \\
(ALTs\ bs\ rs)\backslash c & \stackrel{\text{def}}{=} & ALTs\ bs\ (map\ (\_\backslash c)\ rs) \\
(SEQ\ bs\ r_1\ r_2)\backslash c & \stackrel{\text{def}}{=} & \textit{if } bnullable\ r_1 \\
 & & \textit{then } ALT\ bs\ (SEQ\ []\ (r_1\backslash c)\ r_2) \\
 & & \qquad\qquad (fuse\ (bmkeps\ r_1)\ (r_2\backslash c)) \\
 & & \textit{else } SEQ\ bs\ (r_1\backslash c)\ r_2 \\
(STAR\ bs\ r)\backslash c & \stackrel{\text{def}}{=} & SEQ\ bs\ (fuse\ [Z] (r\backslash c))\ (STAR\ []\ r)
\end{array}
$$

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

$$
\begin{array}{lcl}
blexer\ r\ s & \stackrel{\text{def}}{=} & \textit{let } r_{der} = (r^{\uparrow})\backslash s \textit{ in} \\
 & & \quad \textit{if } bnullable(r_{der}) \ \textit{ then } decode\ (bmkeps\ r_{der})\ r \ \textit{ else } None
\end{array}
$$

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.

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

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 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 3.** *If* $\vdash v : r$ *then* $code\ v = retrieve\ (r^{\uparrow})\ v$.

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 4.**

(1) $(a\backslash s)^{\downarrow} = (a^{\downarrow})\backslash s$

(2) $bnullable(a)$ *iff* $nullable(a^{\downarrow})$

(3) $bmkeps(a) = retrieve\ a\ (mkeps\ (a^{\downarrow}))$ *provided* $nullable(a^{\downarrow})$.

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 $r\backslash s$ 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 $r^{\uparrow}\backslash s$ and then retrieve the corresponding bitcoded version, followed by a decoding step.

▶ **Lemma 5** (Main Lemma). *If* $\vdash v : r\backslash s$ *then*

$$\mathit{Some}\,(\mathit{flex}\,r\,\mathit{id}\,s\,v) = \mathit{decode}(\mathit{retrieve}\,(r^{\uparrow}\backslash s)\,v)\,r$$

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.[1]

The case for $[]$ is routine using Lemmas 2 and 3. In the case $s\,@\,[c]$, we can infer from the assumption that $\vdash v : (r\backslash s)\backslash c$ holds. Hence by Lemma **??** we know that (*) $\vdash \mathit{inj}\,(r\backslash s)\,c\,v : r\backslash s$ holds too. By definition of *flex* we can unfold the left-hand side to be

$$\mathit{Some}\,(\mathit{flex}\;r\;\mathit{id}\,(s\,@\,[c])\,v) = \mathit{Some}\,(\mathit{flex}\;r\;\mathit{id}\,s\,(\mathit{inj}\,(r\backslash s)\,c\,v))$$

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

$$\mathit{decode}\,(\mathit{retrieve}\,(r^{\uparrow}\backslash s)\;(\mathit{inj}\,(r\backslash s)\;c\;v))\,r$$

which is equal to $\mathit{decode}\,(\mathit{retrieve}\,(r^{\uparrow}\backslash(s\,@\,[c]))\,v)\,r$ 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 6.** $\mathit{lexer}\,r\,s = \mathit{blexer}\,r\,s$

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 4*(2)* we know the *if*-tests coincide:

$$\mathit{bnullable}(r^{\uparrow}\backslash s)\;\mathit{iff}\;\mathit{nullable}(r\backslash s)$$

For the *if*-branch suppose $r_d \overset{\mathrm{def}}{=} r^{\uparrow}\backslash s$ and $d \overset{\mathrm{def}}{=} r\backslash s$. We have (*) $\mathit{nullable}\,d$. We can then show by Lemma 4*(3)* that

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

where the right-hand side is equal to $\mathit{Some}\,(\mathit{flex}\,r\,\mathit{id}\,s\,(\mathit{mkeps}\,d))$ by Lemma 5 (we know $\vdash \mathit{mkeps}\,d : d$ 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.

[1] Isabelle/HOL provides an induction principle for this way of performing the induction.

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

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

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

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

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

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

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

Figure 3 ???

4 Simplification

Derivatives as calculated by Brzozowski's method are usually more complex regular expressions than the initial one; the result is that the derivative-based matching and lexing algorithms are often abysmally slow.

However, as Sulzmann and Lu wrote, various optimisations are possible, such as the simplifications of 0 + r,r + 0,1 · r and r · 1 to r. These simplifications can speed up the algorithms considerably.

▶ **Lemma 7.** *If* $r_1 \rightsquigarrow r_2$ *then* $bnullable\ r_1 = bnullable\ r_2$.

▶ **Lemma 8.** *If* $r_1 \rightsquigarrow r_2$ *and* $bnullable\ r_1$ *then* $bmkeps\ r_1 = bmkeps\ r_2$.

▶ **Lemma 9.** $r \rightsquigarrow^* bsimp\ r$

▶ **Lemma 10.** *If* $r_1 \rightsquigarrow r_2$ *then* $r_1\backslash c \rightsquigarrow^* r_2\backslash c$.

▶ **Lemma 11.** $r\backslash s \rightsquigarrow^* r\backslash_{simp}\ s$

▶ **Theorem 12.** $blexer\ r\ s = blexer^{+}\ r\ s$

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.

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.