POSIX Lexing with Bitcoded Derivatives

Chengsong Tan ✉
King's College London

Christian Urban ✉
King's College London

Abstract

Sulzmann and Lu describe 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. This information is needed in the context of lexing in order to extract and to classify tokens. 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 of some 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 and lexing, 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 derivative-based matcher in HOL4 by Owens and Slind [8]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [5]. And another one in Coq is given by Coquand and Siles [4]. Also Ribeiro and Du Bois give one in Agda [9].

However, there are two difficulties with derivative-based matchers: First, Brzozowski's original matcher only generates a yes/no answer for whether a regular expression matches a string or not. This is too little information in the context of lexing where separate tokens must be identified and also classified (for example as keywords or identifiers). Sulzmann and Lu [10] overcome this difficulty by cleverly extending Brzozowski's matching algorithm. Their extended version generates additional information on *how* a regular expression matches a string following the POSIX rules for regular expression matching. They achieve this by adding a second "phase" to Brzozowski's algorithm involving an injection function. In our own earlier work we provided the formal specification of what POSIX matching means and proved in Isabelle/HOL the correctness of Sulzmann and Lu's extended algorithm accordingly [2].

The second difficulty 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

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$$
\begin{array}{lcl}
(a + aa)^* & \xrightarrow{\_\backslash a} & (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* \\
& \xrightarrow{\_\backslash a} & (\mathbf{0} + \mathbf{0}a + \mathbf{1}) \cdot (a + aa)^* \;+\; (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* \\
& \xrightarrow{\_\backslash a} & (\mathbf{0} + \mathbf{0}a + \mathbf{0}) \cdot (a + aa)^* + (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* + \\
& & \quad (\mathbf{0} + \mathbf{0}a + \mathbf{1}) \cdot (a + aa)^* + (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* \\
& \xrightarrow{\_\backslash a} & \ldots \qquad\qquad \text{(regular expressions of sizes 98, 169, 283, 468, 767, \ldots)}
\end{array}
$$

where after around 35 steps we run out of memory on a typical computer (we shall define shortly the precise details of our regular expressions and the derivative operation). 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 simplifications have been proved in our earlier work to preserve the correctness of Sulzmann and Lu's algorithm [2], they unfortunately do *not* help with limiting the growth of the derivatives shown above: the growth is slowed, but the derivatives can still grow quickly beyond any finite bound.

Sulzmann and Lu overcome this "growth problem" in a second algorithm [10] where they introduce bitcoded regular expressions. In this version, POSIX values are represented as bitsequences and such sequences are incrementally generated when derivatives are calculated. The compact representation of bitsequences and regular expressions allows them to define a more "aggressive" simplification method that keeps the size of the derivatives finite no matter what the length of the string is. They make some informal claims about the correctness and linear behaviour of this version, but do not provide any supporting proof arguments, not even "pencil-and-paper" arguments. They write about their bitcoded *incremental parsing method* (that is the algorithm to be formalised in this paper):

> *"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."*

Contributions: We have implemented in Isabelle/HOL the derivative-based lexing algorithm of Sulzmann and Lu [10] where regular expressions are annotated with bitsequences. We define the crucial simplification function as a recursive function, instead of a fix-point operation. One objective of the simplification function is to remove duplicates of regular expressions. For this Sulzmann and Lu use in their paper the standard *nub* function from Haskell's list library, but this function does not achieve the intended objective with bitcoded regular expressions. The reason is that in the bitcoded setting, each copy generally has a different bitcode annotation—so *nub* would never "fire". Inspired by Scala's library for lists, we shall instead use a *distinctBy* function that finds duplicates under an erasing function that deletes bitcodes. We shall also introduce our own argument and definitions for establishing the correctness of the bitcoded algorithm when simplifications are included.

In this paper, we shall first briefly introduce the basic notions of regular expressions and describe the basic definitions of POSIX lexing from our earlier work [2]. This serves as a reference point for what correctness means in our Isabelle/HOL proofs. We shall then prove the correctness for the bitcoded algorithm without simplification, and after that extend the proof to include simplification.

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 $\_ @ \_$. We often 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 regular expressions are defined as usual as the elements of the following inductive datatype:

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

where $\mathbf{0}$ stands for the regular expression that does not match any string, $\mathbf{1}$ for the regular expression that matches only the empty string and c for matching a character literal. The constructors $+$ and $\cdot$ represent alternatives and sequences, respectively. The *language* of a regular expression, written L, is defined as usual and we omit giving the definition here (see for example [2]).

Central to Brzozowski's regular expression matcher are two functions called *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.

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

We can extend this definition to give derivatives w.r.t. strings:

$$r\backslash[] \stackrel{\text{def}}{=} r \qquad r\backslash(c :: s) \stackrel{\text{def}}{=} (r\backslash c)\backslash s$$

Using *nullable* and the derivative operation, we can define the following simple regular expression matcher:

$$\textit{match } s\ r \stackrel{\text{def}}{=} \textit{nullable}(r\backslash s)$$

This is essentially Brzozowski's algorithm from 1964. Its main virtue is that the algorithm can be easily implemented as a functional program (either in a functional programming language or in a theorem prover). The correctness proof for *match* amounts to establishing the property

▶ **Proposition 1.** *match s r* *if and only if* $s \in L(r)$

It is a fun exercise to formaly prove this property in a theorem prover.

The novel idea of Sulzmann and Lu is to extend this algorithm for lexing, where it is important to find out which part of the string is matched by which part of the regular expression. For this Sulzmann and Lu presented two lexing algorithms in their paper [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

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

where we use *vs* to stand for a list of values. The string underlying a value can be calculated by a *flat* function, written $|\_|$. It traverses a value and collects the characters contained in it. Sulzmann and Lu also define inductively an inhabitation relation that associates values to regular expressions:

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

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

$$\frac{\vdash v_1 : r_1 \qquad \vdash v_2 : r_2}{\vdash \textit{Seq } v_1\ v_2 : r_1 \cdot r_2} \qquad \frac{\forall v \in vs. \vdash v : r \wedge |v| \neq []}{\vdash \textit{Stars } vs : r^*}$$

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

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

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

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

Figure 1 The inductive definition of POSIX values taken from our earlier paper [2]. The ternary relation, written $(s, r) \to v$, formalises the notion of given a string s and a regular expression r what is the unique value v that satisfies the informal POSIX constraints for regular expression matching.

Note that no values are associated with the regular expression $\mathbf{0}$, since it cannot match any string. It is routine to establish how values "inhabiting" a regular expression correspond to the language of a regular expression, namely

▶ **Proposition 2.** $L\ r = \{|v| \mid \vdash v : r\}$

In general there is more than one value inhabited by a regular expression (meaning regular expressions can typically match more than one string). But even when fixing a string from the language of the regular expression, there are generally more than one way of how the regular expression can match this string. POSIX lexing is about identifying the unique value for a given regular expression and a string that satisfies the informal POSIX rules (see [1, 6, 7, 10, 11]).[1] Sometimes these informal rules are called *maximal much rule* and *rule priority*. One contribution of our earlier paper is to give a convenient specification for what a POSIX value is (the inductive rules are shown in Figure 1).

The clever idea by Sulzmann and Lu [10] in their first algorithm is to define an injection function on values that mirrors (but inverts) the construction of the derivative on regular expressions. Essentially it injects back a character into a value. For this they define two functions called *mkeps* and *inj*:

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

$$\begin{array}{lcl}
inj\ d\ c\ (Empty) & \stackrel{\text{def}}{=} & Char\ d \\
inj\ (r_1 + r_2)\ c\ (Left\ v_1) & \stackrel{\text{def}}{=} & Left\ (inj\ r_1\ c\ v_1) \\
inj\ (r_1 + r_2)\ c\ (Right\ v_2) & \stackrel{\text{def}}{=} & Right\ (inj\ r_2\ c\ v_2) \\
inj\ (r_1 \cdot r_2)\ c\ (Seq\ v_1\ v_2) & \stackrel{\text{def}}{=} & Seq\ (inj\ r_1\ c\ v_1)\ v_2 \\
inj\ (r_1 \cdot r_2)\ c\ (Left\ (Seq\ v_1\ v_2)) & \stackrel{\text{def}}{=} & Seq\ (inj\ r_1\ c\ v_1)\ v_2 \\
inj\ (r_1 \cdot r_2)\ c\ (Right\ v_2) & \stackrel{\text{def}}{=} & Seq\ (mkeps\ r_1)\ (inj\ r_2\ c\ v_2) \\
inj\ (r^*)\ c\ (Seq\ v\ (Stars\ vs)) & \stackrel{\text{def}}{=} & Stars\ (inj\ r\ c\ v :: vs)
\end{array}$$

[1] POSIX lexing acquired its name from the fact that the corresponding rules were described as part of the POSIX specification for Unix-like operating systems [1].

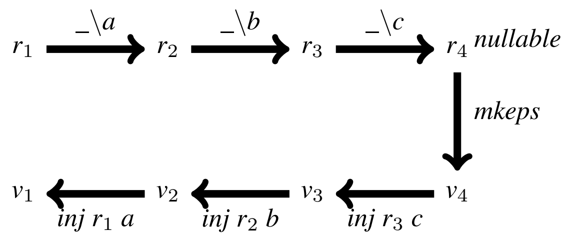

Figure 2 The two phases of the first algorithm by Sulzmann & Lu [10], matching the string $[a, b, c]$. The first phase (the arrows from left to right) is Brzozowski's matcher building successive derivatives. If the last regular expression is *nullable*, then the functions of the second phase are called (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. The value v_1 is the result of the algorithm representing the POSIX value for this string and regular expression.

The function *mkeps* is called when the last derivative is nullable, that is the string to be matched is in the language of the regular expression. It generates a value for how the last derivative can match the empty string. The injection function then calculates the corresponding value for each intermediate derivative until a value for the original regular expression is generated. Graphically the algorithm by Sulzmann and Lu can be illustrated by the picture in Figure 2 where the path from the left to the right involving *derivatives*/*nullable* is the first phase of the algorithm (calculating successive Brzozowski's derivatives) and *mkeps*/*inj*, the path from right to left, the second phase. This picture shows the steps required when a regular expression, say r_1, matches the string $[a, b, c]$. The lexing algorithm by Sulzmann and Lu can be defined as:

$$\begin{array}{lcl}
\textit{lexer}\ r\ [] & \stackrel{\text{def}}{=} & \textit{if nullable}\ r\ \textit{then Some}\ (\textit{mkeps}\ r)\ \textit{else None}\\
\textit{lexer}\ r\ (c :: s) & \stackrel{\text{def}}{=} & \textit{case lexer}\ (r\backslash c)\ s\ \textit{of}\\
 & & \quad \textit{None} \Rightarrow \textit{None}\\
 & & \mid \textit{Some}\ v \Rightarrow \textit{Some}\ (\textit{inj}\ r\ c\ v)
\end{array}$$

We have shown in our earlier paper [2] that the algorithm by Sulzmann and Lu is correct. The cenral property we established relates the derivative operation to the injection function.

▶ **Proposition 3.** *If* $(s,\ r\backslash c) \rightarrow v$ *then* $(c :: s,\ r) \rightarrow \textit{inj}\ r\ c\ v$.

With this in place we were able to prove:

▶ **Proposition 4.**

(1) $s \notin L\ r$ *if and only if* $\textit{lexer}\ r\ s = \textit{None}$

(2) $s \in L\ r$ *if and only if* $\exists\, v.\ \textit{lexer}\ r\ s = \textit{Some}\ v \wedge (s, r) \rightarrow v$

In fact we have shown that in the success case the generated POSIX value v is unique and in the failure case that there is no POSIX value v that satisfies $(s, r) \rightarrow v$. While the algorithm is correct, it is excrutiatingly slow in examples where the derivatives grow arbitrarily (see example from the Introduction). However it can be used as a conveninet reference point for the correctness proof of the second algorithm by Sulzmann and Lu, which we shall describe next.

3 Bitcoded Regular Expressions and Derivatives

In the second part of their paper [10], Sulzmann and Lu describe another algorithm that also 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

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

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 $ALT\ bs\ r_1\ r_2$ 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. For this Sulzmann and Lu 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). 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*:

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

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 5.** *If* $\vdash v : r$ *then* $decode\ (code\ v)\ r = Some\ v$.

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

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

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

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

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.

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

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

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.

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

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*:

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

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.

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

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 6.** *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 7.**
(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. ◀

The only problem left for the correctness proof is that the bitcoded algorithm has only a "forward phase" where POSIX values are generated incrementally. We can achive the same effect with *lexer* by stacking up injection functions in the during forward phase. An auxiliary function, called *flex*, allows us to recast the rules of $lexer$ (with its two phases) in terms of a single phase.

$$
\begin{array}{lcl}
\textit{flex}\;r\;f\;[] & \stackrel{\text{def}}{=} & f\\
\textit{flex}\;r\;f\;(c::s) & \stackrel{\text{def}}{=} & \textit{flex}\;(r\backslash c)\;(\lambda v.\;f\;(\textit{inj}\;r\;c\;v))\;s
\end{array}
$$

The point of this function is that when reaching the end of the string, we just need to apply the stacked injection functions to the value generated by $mkeps$. Using this function we can recast the success case in *lexer* as follows:

▶ **Proposition 8.** *If lexer* $r\;s$ = *Some* v *then* v = *flex* r *id* $s\;(mkeps(r\backslash s))$.

Note we did not redefine *lexer*, we just established that the value generated by *lexer* can also be obtained by a different method. While this different method is not efficient (we essentially need to traverse the string s twice, once for building the derivative $r\backslash s$ and another time for stacking up injection functions using *flex*), it helped us with proving that incrementally building up values.

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

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

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.[2]

The case for $[]$ is routine using Lemmas 5 and 6. In the case $s\,@\,[c]$, we can infer from the assumption that $\vdash v : (r\backslash s)\backslash c$ holds. Hence by Prop. 3 we know that (*) $\vdash \textit{inj}\,(r\backslash s)\,c\,v : r\backslash s$ holds too. By definition of *flex* we can unfold the left-hand side to be

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

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

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

which is equal to $\textit{decode}\,(\textit{retrieve}\,(r^{\uparrow}\backslash(s\,@\,[c]))\,v)\,r$ 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 10.** $\textit{lexer}\,r\,s = \textit{blexer}\,r\,s$

Proof. We can first expand both sides using Prop. 8 and the definition of *blexer*. This gives us two *if*-statements, which we need to show to be equal. By Lemma 7*(2)* we know the *if*-tests coincide:

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

For the *if*-branch suppose $r_d \stackrel{\text{def}}{=} r^{\uparrow}\backslash s$ and $d \stackrel{\text{def}}{=} r\backslash s$. We have (*) $\textit{nullable}\,d$. We can then show by Lemma 7*(3)* that

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

where the right-hand side is equal to $\textit{Some}\,(\textit{flex}\,r\,\textit{id}\,s\,(\textit{mkeps}\,d))$ by Lemma 9 (we know $\vdash \textit{mkeps}\,d : d$ 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.

[2] 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 $\mathbf{0}\,r \Rightarrow \mathbf{0}$, $\mathbf{1}\,r \Rightarrow r$, $\mathbf{0} + r \Rightarrow r$ and $r + r \Rightarrow r$. While these simplifications can speed up the algorithms considerably in many cases, they do not solve fundamentally the "growth problem" with derivatives. To see this let us return to the example

▶ **Lemma 11.** *If* $r_1 \rightsquigarrow r_2$ *then* $bnullable\ r_1 = bnullable\ r_2$.

▶ **Lemma 12.** *If* $r_1 \rightsquigarrow r_2$ *and* $bnullable\ r_1$ *then* $bmkeps\ r_1 = bmkeps\ r_2$.

▶ **Lemma 13.** $r \rightsquigarrow^* bsimp\ r$

▶ **Lemma 14.** *If* $r_1 \rightsquigarrow r_2$ *then* $r_1 \backslash c \rightsquigarrow^* r_2 \backslash c$.

▶ **Lemma 15.** $r \backslash s \rightsquigarrow^* r \backslash_{simp}\ s$

▶ **Theorem 16.** $blexer\ r\ s = blexer^{+}\ r\ s$

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

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

7 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.

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

9 R. Ribeiro and A. Du Bois. Certified Bit-Coded Regular Expression Parsing. In *Proc. of the 21st Brazilian Symposium on Programming Languages*, New York, NY, USA, 2017. Association for Computing Machinery.

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.