# 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 variant 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

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

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 **0**s and **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 rather 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]. The 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:

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

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

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

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

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

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

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

**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 **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*:

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

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

[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].

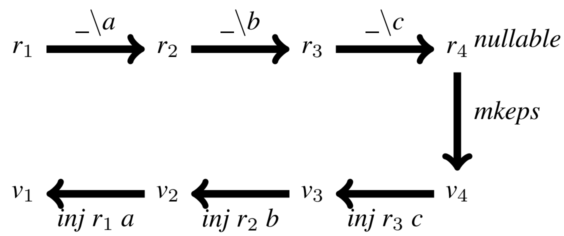


**Figure 2** The two phases of the first algorithm by Sulzmann & Lu [10], matching the string $[a, b, c]$. The first phase (the arrows from left to right) is Brzozowski's matcher building successive derivatives. If the last regular expression is *nullable*, then the functions of the second phase are 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 run 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. The picture above 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 this algorithm is correct, that is it generates POSIX values. The cenral property we established relates the derivative operation to the injection function.

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

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* $\exists v.\ \textit{lexer}\ r\ s = \textit{Some}\ v \wedge (s, r) \to v$

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 $(s, r) \to v$. While the algorithm is correct, it is excrutiatingly slow in cases where the derivatives grow arbitrarily (see example from the Introduction). However it can be used as a convenient 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

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

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 $\textit{ALT}\ bs\ r_1\ r_2$ is just an abbreviation for $\textit{ALTs}\ bs\ [r_1, r_2]$. 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.

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

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 $\textit{decode}'$ and *decode*:

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

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

**Proof.** This follows from the property that $\textit{decode}'\ ((\textit{code}\ v)\ @\ bs)\ r = (v, bs)$ 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 $r^\downarrow$, 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.

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

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

The key function in the bitcoded algorithm is the derivative of a 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.

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

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*:

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

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 *lexer*.

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

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

The idea behind this function is to retrieve a possibly partial bitsequence 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* $v = \textit{retrieve}\,(r^{\uparrow})\,v$.

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

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

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

The only difficulty left for the correctness proof is that the bitcoded algorithm has only a "forward phase" where POSIX values are generated incrementally. We can achieve the same effect with *lexer* by stacking up injection functions during the 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 and stacked up injection functions.

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

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* $\textit{lexer}\,r\,s = \textit{Some}\ v$ *then* $v = \textit{flex}\,r\,\textit{id}\,s\,(\textit{mkeps}(r\backslash s))$.

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 helps us with proving that incrementally building up values generates the same result.

This brings us to our main lemma in this section: if we calculate a derivative, say $r\backslash s$ and have a value, say $v$, inhabited by this derivative, then we can produce the result $\textit{lexer}$ generates by applying this value to the stacked-up injection functions that *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*—it indeed 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 (*) *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 by Sulzmann and Lu 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.

## 4 Simplification

Derivatives as calculated by Brzozowski's method are usually more complex regular expressions than the initial one; the result is that derivative-based matching and lexing algorithms are often abysmally slow if the "growth problem" is not addressed. 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 considerably speed up the two algorithms in many cases, they do not solve fundamentally the growth problem with derivatives. To see this let us return to the example from the Introduction that shows the derivatives for $(a + aa)^*$. If we delete in the 3rd step all $\mathbf{0}s$ and $\mathbf{1}$s according to the simplification rules shown above we obtain

$$(a + aa)^* \quad \xrightarrow{\_\backslash[a,a,a]} \quad \underbrace{(\mathbf{1} + \mathbf{1}a)\cdot(a + aa)^*}_{r} + ((a + aa)^* + \underbrace{(\mathbf{1} + \mathbf{1}a)\cdot(a + aa)^*}_{r}) \qquad (1)$$

This is a simpler derivative, but unfortunately we cannot make further simplifications. This is a problem because the outermost alternatives contains two copies of the same regular expression (underlined with $r$). The copies will spawn new copies in later steps and they in turn further copies. This destroys an hope of taming the size of the derivatives. But the second copy of $r$ in (1) will never contribute to a value, because POSIX lexing will always prefer matching a string with the first copy. So in principle it could be removed. The dilemma with the simple-minded simplification rules above is that the rule $r + r \Rightarrow r$ will never be applicable because as can be seen in this example the regular expressions are separated by another sub-regular expression.

But here is where Sulzmann and Lu's representation of generalised alternatives in the bitcoded algorithm shines: in *ALTs bs rs* we can define a more aggressive simplification by recursively simplifying all regular expressions in *rs* and then analyse the resulting list and remove any duplicates. Another advantage is that the bitsequences in bitcoded regular expressions can be easily modified such that simplification does not interfere with the value constructions. For example we can "flatten", or de-nest, *ALTs* as follows

$$\mathit{ALTs}\ \mathit{bs}_1\ (\mathit{ALTs}\ \mathit{bs}_2\ \mathit{rs}_2 :: \mathit{rs}_1) \quad \xrightarrow{\mathit{bsimp}} \quad \mathit{ALTs}\ \mathit{bs}_1\ (\mathit{map}\ (\mathit{fuse}\ \mathit{bs}_2)\ \mathit{rs}_2 :: \mathit{rs}_1)$$

where we just need to fuse the bitsequence that has accumulated in $\mathit{bs}_2$ to the alternatives in $\mathit{rs}_2$. As we shall show below this will ensure that the correct value corresponding to the original (unsimplified) regular expression can still be extracted.

However there is one problem with the definition for the more aggressive simlification rules by Sulzmann and Lu. Recasting their definition with our syntax they define the step of removing duplicates as

$$\mathit{bsimp}\ (\mathit{ALTs}\ \mathit{bs}\ \mathit{rs}) \stackrel{\text{def}}{=} \mathit{ALTs}\ \mathit{bs}\ (\mathit{nup}\ (\mathit{map}\ \mathit{bsimp}\ \mathit{rs}))$$

where they first recursively simplify the regular expressions in *rs* (using *map*) and then use Haskell's *nub*-function to remove potential duplicates. While this makes sense when considering the example shown in (1), *nub* is the inappropriate function in the case of bitcoded regular expressions. The reason is that in general the n elements in *rs* will have a different bitsequence annotated to it and in this way *nub* will never find a duplicate to be removed. The correct way to handle this situation is to first *erase* the regular expressions when comparing potential duplicates. This is inspired by Scala's list functions of the form *distinctBy rs f acc* where a function is applied first before two elements are compared. We define this function in Isabelle/HOL as

$$\begin{aligned}
\mathit{distinctBy}\ []\ f\ \mathit{acc} &\stackrel{\text{def}}{=} []\\
\mathit{distinctBy}\ (x :: \mathit{xs})\ f\ \mathit{acc} &\stackrel{\text{def}}{=} \mathit{if}\ f\ x \in \mathit{acc}\ \mathit{then}\ \mathit{distinctBy}\ \mathit{xs}\ f\ \mathit{acc}\ \mathit{else}\ x :: \mathit{distinctBy}\ \mathit{xs}\ f\ (\{f\ x\} \cup \mathit{acc})
\end{aligned}$$

where we scan the list from left to right (because we have to remove later copies). In this function, $f$ is a function and $acc$ is an accumulator for regular expressions—essentially a set of elements we have already seen while scanning the list. Therefore we delete an element, say $x$, from the list provided $f\,x$ is already in the accumulator; otherwise we keep $x$ and scan the rest of the list but now add $f\,x$ as another element to $acc$. We will use *distinctBy* where $f$ is our erase functions, $\_^{\downarrow}$, that deletes bitsequences from bitcoded regular expressions. This is clearly a computationally more expensive operation, than *nub*, but is needed in order to make the removal of unnecessary copies to work.

Our simplification function depends on three helper functions, one is called *flts* and defined as follows:

$$\begin{array}{lcl} \mathit{flts}\ [] & \stackrel{\text{def}}{=} & [] \\ \mathit{flts}\ (\mathit{ZERO} :: \mathit{rs}) & \stackrel{\text{def}}{=} & \mathit{flts}\ \mathit{rs} \\ \mathit{flts}\ (\mathit{ALTs}\ \mathit{bs}'\ \mathit{rs}' :: \mathit{rs}) & \stackrel{\text{def}}{=} & \mathit{map}\ (\mathit{fuse}\ \mathit{bs}')\ \mathit{rs}'\ @\ \mathit{flts}\ \mathit{rs} \end{array}$$

The second clause removes all instances of *ZERO* in alternatives and the second "spills" out nested alternatives (but retaining the bitsequence $bs'$ accumulated in the inner alternative). There are some corner cases to be considered when the resulting list inside an alternative is empty or a singleton list. We take care of those cases in the *bsimpALTs* function; similarly we define a helper function that simplifies sequences according to the usual rules about *ZERO*s and *ONE*s:

$$\begin{array}{lcl} \mathit{bsimpALTs}\ \mathit{bs}\ [] & \stackrel{\text{def}}{=} & \mathit{ZERO} \\ \mathit{bsimpALTs}\ \mathit{bs}\ [r] & \stackrel{\text{def}}{=} & \mathit{fuse}\ \mathit{bs}\ r \\ \mathit{bsimpALTs}\ \mathit{bs}\ \mathit{rs} & \stackrel{\text{def}}{=} & \mathit{ALTs}\ \mathit{bs}\ \mathit{rs} \end{array} \qquad \begin{array}{lcl} \mathit{bsimpSEQ}\ \mathit{bs}\ \_\ \mathit{ZERO} & \stackrel{\text{def}}{=} & \mathit{ZERO} \\ \mathit{bsimpSEQ}\ \mathit{bs}\ \mathit{ZERO}\ \_ & \stackrel{\text{def}}{=} & \mathit{ZERO} \\ \mathit{bsimpSEQ}\ \mathit{bs}_1\ (\mathit{ONE}\ \mathit{bs}_2)\ r_2 & \stackrel{\text{def}}{=} & \mathit{fuse}\ (\mathit{bs}_1\ @\ \mathit{bs}_2)\ r_2 \\ \mathit{bsimpSEQ}\ \mathit{bs}\ r_1\ r_2 & \stackrel{\text{def}}{=} & \mathit{SEQ}\ \mathit{bs}\ r_1\ r_2 \end{array}$$

With this in place we can define our simlification function as

$$\begin{array}{lcl} \mathit{bsimp}\ (\mathit{SEQ}\ \mathit{bs}\ r_1\ r_2) & \stackrel{\text{def}}{=} & \mathit{bsimpSEQ}\ \mathit{bs}\ (\mathit{bsimp}\ r_1)\ (\mathit{bsimp}\ r_2) \\ \mathit{bsimp}\ (\mathit{ALTs}\ \mathit{bs}\ \mathit{rs}) & \stackrel{\text{def}}{=} & \mathit{bsimpALT}\ \mathit{bs}\ (\mathit{distinctBy}\ (\mathit{flts}\ (\mathit{map}\ \mathit{bsimp}\ \mathit{rs}))\ \mathit{erase}\ \varnothing) \\ \mathit{bsimp}\ r & \stackrel{\text{def}}{=} & r \end{array}$$

As far as we can see, our recursive function *bsimp* simplifies regular expressions as intended by Sulzmann and Lu. There is no point to apply the *bsimp* function repeatedly (like the simplification in their paper which is applied until a fixpoint is reached), because we can show that it is idempotent, that is

▶ **Proposition 11.** *???*

▶ **Lemma 12.** *If* $r_1 \leadsto r_2$ *then* $\mathit{bnullable}\ r_1 = \mathit{bnullable}\ r_2$.

▶ **Lemma 13.** *If* $r_1 \leadsto r_2$ *and* $\mathit{bnullable}\ r_1$ *then* $\mathit{bmkeps}\ r_1 = \mathit{bmkeps}\ r_2$.

▶ **Lemma 14.** $r \leadsto^* \mathit{bsimp}\ r$

▶ **Lemma 15.** *If* $r_1 \leadsto r_2$ *then* $r_1\backslash c \leadsto^* r_2\backslash c$.

▶ **Lemma 16.** $r\backslash s \leadsto^* r\backslash_{simp}\, s$

▶ **Theorem 17.** $\mathit{blexer}\ r\ s = \mathit{blexer}^+\ r\ s$

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

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

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

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

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

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

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

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

**Figure 3** ???

# 5 Bound - NO

# 6 Conclusion

## 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.