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 finitely bounded. 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 finite bounds 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

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. 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*). The beauty of Brzozowski's derivatives [5] 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. Another attractive feature of derivatives is that they can be easily extended to *bounded* regular expressions, such as $r^{\{n\}}$ or $r^{\{..n\}}$, where numbers or intervals of numbers specify how many times a regular expression should be used during matching.

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 [14] 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 [3].

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

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

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 [3], they unfortunately do *not* help with limiting the growth of the derivatives shown above: the growth is slowed, but some derivatives can still grow rather quickly beyond any finite bound.

Sulzmann and Lu address this "growth problem" in a second algorithm [14] 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 finitely bounded 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." [14, Page 14]*

Contributions: We have implemented in Isabelle/HOL the derivative-based lexing algorithm of Sulzmann and Lu [14] where regular expressions are annotated with bitsequences. We define the crucial simplification function as a recursive function, without the need of a fixpoint 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 *distinctWith* function that finds duplicates under an "erasing" function which deletes bitcodes before comparing regular expressions. We shall also introduce our *own* argument and definitions for establishing the correctness of the bitcoded algorithm when simplifications are included. Finally we establish that the size of derivatives can be finitely bounded.

In this paper, we shall first briefly introduce the basic notions of regular expressions and describe the definition of POSIX lexing from our earlier work [3]. 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 the following inductive datatype:

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

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. We sometimes omit the $\cdot$ in a sequence regular expression for brevity. The *language* of a regular expression, written $L(r)$, is defined as usual and we omit giving the definition here (see for example [3]).

In our work here we also add to the usual "basic" regular expressions the *bounded* regular expression $r^{\{n\}}$ where the n specifies that r should match exactly n-times. Again for brevity we omit the other bounded regular expressions $r^{\{..n\}}$, $r^{\{n..\}}$ and $r^{\{n..m\}}$ which specify intervals for how many times r should match. The results presented in this paper extend straightforwardly to them too. The importance of the bounded regular expressions is that they are often used in practical applications, such as Snort (a system for detecting network intrusion) and also in XML Schema definitions. According to Björklund et al [4], bounded regular expressions occur frequently in the latter and can have counters of up to ten million. The problem is that tools based on the classic notion of automata need to expand $r^{\{n\}}$ into n connected copies of the automaton for r. This leads to very inefficient matching algorithms or algorithms that consume large amounts of memory. A classic example is the regular expression $(a+b)^* \cdot a \cdot (a+b)^{\{n\}}$ where the minimal DFA requires at least 2^{n+1} states (see [15]). Therefore regular expression matching libraries that rely on the classic notion of DFAs often impose adhoc limits for bounded regular expressions: For example in the regular expression matching library in the Go language and also in Google's RE2 library the regular expression $a^{\{\mathit{1001}\}}$ is not permitted, because no counter can be above 1000; and in the built-in regular expression library in Rust expressions such as $a^{\{\mathit{1000}\}\{\mathit{100}\}\{\mathit{5}\}}$ give an error message for being too big. These problems can of course be solved in matching algorithms where automata go beyond the classic notion and for instance include explicit counters (see [15]). The point here is that Brzozowski derivatives and the algorithms by Sulzmann and Lu can be straightforwardly extended to deal with bounded regular expressions and moreover the resulting code still consists of only simple recursive functions and inductive datatypes. Finally, bounded regular expressions do not destroy our finite boundedness property, which we shall prove later on.

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.

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

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

We can extend this definition to give derivatives w.r.t. strings, namely $r\backslash[] \stackrel{\text{def}}{=} r$ and $r\backslash(c::s) \stackrel{\text{def}}{=} (r\backslash c)\backslash s$. Using *nullable* and the derivative operation, we can define a simple regular expression matcher, namely $\mathit{match}\ s\ r \stackrel{\text{def}}{=} \mathit{nullable}(r\backslash s)$. 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 of *match* amounts to establishing the property:

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

It is a fun exercise to formally prove this property in a theorem prover. We are aware of a mechanised correctness proof of Brzozowski's derivative-based matcher in HOL4 by Owens and Slind [12]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [9]. And another one in Coq is given by Coquand and Siles [6]. Also Ribeiro and Du Bois give one in Agda [13].

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 [14]. 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

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

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 [3].

Sulzmann and Lu also define inductively an inhabitation relation that associates values to regular expressions. Our version of this relation is defined the following six rules for the values:

$$\frac{}{\vdash \mathit{Empty} : \mathbf{1}} \qquad \frac{}{\vdash \mathit{Char}\ c : c} \qquad \frac{\vdash v_1 : r_1}{\vdash \mathit{Left}\ v_1 : r_1 + r_2} \qquad \frac{\vdash v_2 : r_2}{\vdash \mathit{Right}\ v_2 : r_1 + r_2} \qquad \frac{\vdash v_1 : r_1 \qquad \vdash v_2 : r_2}{\vdash \mathit{Seq}\ v_1\ v_2 : r_1 \cdot r_2}$$

$$\frac{\forall v \in vs.\ \vdash v : r \land |v| \neq []}{\vdash \mathit{Stars}\ vs : r^*} \qquad \frac{\begin{array}{l}\forall v \in vs_1.\ \vdash v : r \land |v| \neq [] \\ \forall v \in vs_2.\ \vdash v : r \land |v| = [] \quad \mathit{len}\ (vs_1\ @\ vs_2) = n\end{array}}{\vdash \mathit{Stars}\ (vs_1\ @\ vs_2) : r^{\{n\}}}$$

Note that no values are associated with the regular expression $\mathbf{0}$, since it cannot match any string. Interesting is our version of the rule for r^* where we require that each value in vs flattens to a non-empty string. This means if r^* "fires" one or more times, then each copy needs to match a non-empty string. Similarly, in the rule for $r^{\{n\}}$ we require that the length of the list $vs_1\ @\ vs_2$ equals n (meaning the regular expression r matches n-times) and that the first segment of this list contains values that flatten to non-empty strings followed by a segment that only contains values that flatten to the empty string. It is routine to establish how values "inhabiting" a regular expression correspond to the language of a regular expression, namely $L\ r = \{|v| \mid \vdash v : r\}$.

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, 10, 11, 14, 16]). Sometimes these informal rules are called *maximal munch rule* and *rule priority*. One contribution of our earlier paper is to give a convenient specification for what POSIX values are (the inductive rules are shown in Figure 1).

The clever idea by Sulzmann and Lu [14] 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*:

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

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

$$\frac{}{([], \mathbf{1}) \to Empty}P\mathbf{1} \qquad \frac{}{([c], c) \to Char\ c}Pc \qquad \frac{(s, r_1) \to v}{(s, r_1 + r_2) \to Left\ v}P{+}L \qquad \frac{(s, r_2) \to v \qquad s \notin L\ r_1}{(s, r_1 + r_2) \to Right\ v}P{+}R$$

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

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

$$\frac{\forall v \in vs.\ ([], r) \to v \qquad len\ vs = n}{([], r^{\{n\}}) \to Stars\ vs}Pn[] \qquad \frac{\begin{array}{c}(s_1, r) \to v \qquad (s_2, r^{\{n\}}) \to Stars\ vs \qquad |v| \neq []\\ \nexists s_3\ s_4.\ s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L\ r \wedge s_4 \in L\ (r^{\{n\}})\end{array}}{(s_1 @ s_2, r^{\{n+1\}}) \to Stars\ (v :: vs)}Pn{+}$$

Figure 1 The inductive definition of POSIX values taken from our earlier paper [3]. 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.

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. In case of $r^{\{n\}}$ we use the function *replicate* in order to generate a list of exactly n copies, which is the length of the list we expect in this case. 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 following picture 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.

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

The picture shows the steps required when a regular expression, say r_1, matches the string $[a, b, c]$. The first lexing algorithm by Sulzmann and Lu can be defined as:

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

We have shown in our earlier paper [3] that this algorithm is correct, that is it generates POSIX values. The central property we established relates the derivative operation to the injection function.

▶ **Proposition 2.** *If* $(s,\ r\backslash c) \to v$ *then* $(c :: s,\ r) \to inj\ r\ c\ v$.

With this in place we were able to prove:

▶ **Proposition 3.** *(1)* $s \notin L\ r$ *if and only if* $lexer\ r\ s = None$.
(2) $s \in L\ r$ *if and only if* $\exists v.\ lexer\ r\ s = Some\ v \wedge (s, r) \to v$.

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 excruciatingly slow in cases where the derivatives grow arbitrarily (recall the 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 [14], Sulzmann and Lu describe another algorithm that also generates POSIX values but dispenses 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

$$\mathit{breg} \;::=\; \mathit{ZERO} \;\mid\; \mathit{ONE}\;\mathit{bs} \;\mid\; \mathit{CHAR}\;\mathit{bs}\;c \;\mid\; \mathit{ALTs}\;\mathit{bs}\;\mathit{rs} \;\mid\; \mathit{SEQ}\;\mathit{bs}\;r_1\;r_2 \;\mid\; \mathit{STAR}\;\mathit{bs}\;r \;\mid\; \mathit{NT}\;\mathit{bs}\;r\;n$$

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 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.

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

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 (see Fig. 2). 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 4.** *If* $\vdash v : r$ *then* $\mathit{decode}\;(\mathit{code}\;v)\;r = \mathit{Some}\;v$.

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.

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

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

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

$$
\begin{array}{lcl}
\textit{decode}'\;\textit{bs}\;(\mathbf{1}) & \stackrel{\text{def}}{=} & (\textit{Empty}, \textit{bs}) \qquad\qquad \textit{decode}'\;\textit{bs}\;(c) \;\stackrel{\text{def}}{=}\; (\textit{Char}\;c, \textit{bs})\\
\textit{decode}'\;(Z :: \textit{bs})\;(r_1 + r_2) & \stackrel{\text{def}}{=} & \textit{let}\;(v, \textit{bs}_1) = \textit{decode}'\;\textit{bs}\;r_1\;\textit{in}\;(\textit{Left}\;v, \textit{bs}_1)\\
\textit{decode}'\;(S :: \textit{bs})\;(r_1 + r_2) & \stackrel{\text{def}}{=} & \textit{let}\;(v, \textit{bs}_1) = \textit{decode}'\;\textit{bs}\;r_2\;\textit{in}\;(\textit{Right}\;v, \textit{bs}_1)\\
\textit{decode}'\;\textit{bs}\;(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \textit{let}\;(v_1, \textit{bs}_1) = \textit{decode}'\;\textit{bs}\;r_1\;\textit{in}\\
& & \textit{let}\;(v_2, \textit{bs}_2) = \textit{decode}'\;\textit{bs}_1\;r_2\quad\textit{in}\;(\textit{Seq}\;v_1\;v_2, \textit{bs}_2)\\
\textit{decode}'\;(S :: \textit{bs})\;(r^*) & \stackrel{\text{def}}{=} & (\textit{Stars}\;[], \textit{bs})\\
\textit{decode}'\;(Z :: \textit{bs})\;(r^*) & \stackrel{\text{def}}{=} & \textit{let}\;(v, \textit{bs}_1) = \textit{decode}'\;\textit{bs}\;r\;\textit{in}\\
& & \textit{let}\;(\textit{Stars}\;\textit{vs}, \textit{bs}_2) = \textit{decode}'\;\textit{bs}_1\;r^*\quad\textit{in}\;(\textit{Stars}\;v :: \textit{vs}, \textit{bs}_2)\\
\textit{decode}'\;(S :: \textit{bs})\;(r^{\{n\}}) & \stackrel{\text{def}}{=} & (\textit{Stars}\;[], \textit{bs})\\
\textit{decode}'\;(Z :: \textit{bs})\;(r^{\{n\}}) & \stackrel{\text{def}}{=} & \textit{let}\;(v, \textit{bs}_1) = \textit{decode}'\;\textit{bs}\;r\;\textit{in}\\
& & \textit{let}\;(\textit{Stars}\;\textit{vs}, \textit{bs}_2) = \textit{decode}'\;\textit{bs}_1\;r^{\{n-1\}}\quad\textit{in}\;(\textit{Stars}\;v :: \textit{vs}, \textit{bs}_2)
\end{array}
$$

$$
\textit{decode}\;\textit{bs}\;r \stackrel{\text{def}}{=} \textit{let}\;(v, \textit{bs}') = \textit{decode}'\;\textit{bs}\;r\;\textit{in}\quad\textit{if}\;\textit{bs}' = []\;\textit{then}\;\textit{Some}\;v\;\textit{else}\;\textit{None}
$$

Figure 2 Two functions, called *decode*′ and *decode*, for decoding a value from a bitsequence with the help of a regular expression.

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*(*s*), which are the “lifted” versions of *nullable* and *mkeps* acting on bitcoded regular expressions.

$$
\begin{array}{lcl}
\textit{bnullable}\;(\textit{ZERO}) & \stackrel{\text{def}}{=} & \textit{False}\\
\textit{bnullable}\;(\textit{ONE}\;\textit{bs}) & \stackrel{\text{def}}{=} & \textit{True}\\
\textit{bnullable}\;(\textit{CHAR}\;\textit{bs}\;c) & \stackrel{\text{def}}{=} & \textit{False}\\
\textit{bnullable}\;(\textit{ALTs}\;\textit{bs}\;\textit{rs}) & \stackrel{\text{def}}{=} & \exists r \in \textit{rs}.\,\textit{bnullable}\;r\\
\textit{bnullable}\;(\textit{SEQ}\;\textit{bs}\;r_1\;r_2) & \stackrel{\text{def}}{=} & \textit{bnullable}\;r_1 \wedge \textit{bnullable}\;r_2\\
\textit{bnullable}\;(\textit{STAR}\;\textit{bs}\;r) & \stackrel{\text{def}}{=} & \textit{True}\\
\textit{bnullable}\;(\textit{NT}\;\textit{bs}\;r\;n) & \stackrel{\text{def}}{=} & \\
\quad \textit{if}\;n = 0\;\textit{then}\;\textit{True}\;\textit{else}\;\textit{bnullable}\;r & &
\end{array}
$$

$$
\begin{array}{lcl}
\textit{bmkeps}\;(\textit{ONE}\;\textit{bs}) & \stackrel{\text{def}}{=} & \textit{bs}\\
\textit{bmkeps}\;(\textit{ALTs}\;\textit{bs}\;\textit{rs}) & \stackrel{\text{def}}{=} & \textit{bs}\;@\;\textit{bmkepss}\;\textit{rs}\\
\textit{bmkeps}\;(\textit{SEQ}\;\textit{bs}\;r_1\;r_2) & \stackrel{\text{def}}{=} & \\
\quad \textit{bs}\;@\;\textit{bmkeps}\;r_1\;@\;\textit{bmkeps}\;r_2 & & \\
\textit{bmkeps}\;(\textit{STAR}\;\textit{bs}\;r) & \stackrel{\text{def}}{=} & \textit{bs}\;@\;[S]\\
\textit{bmkeps}\;(\textit{NT}\;\textit{bs}\;r\;n) & \stackrel{\text{def}}{=} & \\
\quad \textit{if}\;n = 0\;\textit{then}\;\textit{bs}\;@\;[S] & & \\
\quad \textit{else}\;\textit{bs}\;@\;[Z]\;@\;\textit{bmkeps}\;r\;@\;\textit{bmkeps}\;(\textit{NT}\;[]\;r\;(n-1)) & & \\
\textit{bmkepss}\;(r :: \textit{rs}) & \stackrel{\text{def}}{=} & \\
\quad \textit{if}\;\textit{bnullable}\;r\;\textit{then}\;\textit{bmkeps}\;r\;\;\textit{else}\;\textit{bmkepss}\;\textit{rs} & &
\end{array}
$$

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

$$
\begin{array}{lcl}
(\textit{ZERO})\backslash c & \stackrel{\text{def}}{=} & \textit{ZERO}\\
(\textit{ONE}\;\textit{bs})\backslash c & \stackrel{\text{def}}{=} & \textit{ZERO}\\
(\textit{CHAR}\;\textit{bs}\;d)\backslash c & \stackrel{\text{def}}{=} & \textit{if}\;c = d\;\;\textit{then}\;\textit{ONE}\;\textit{bs}\;\textit{else}\;\textit{ZERO}\\
(\textit{ALTs}\;\textit{bs}\;\textit{rs})\backslash c & \stackrel{\text{def}}{=} & \textit{ALTs}\;\textit{bs}\;(\textit{map}\;(\_\backslash c)\;\textit{rs})\\
(\textit{SEQ}\;\textit{bs}\;r_1\;r_2)\backslash c & \stackrel{\text{def}}{=} & \textit{if}\;\textit{bnullable}\;r_1\\
& & \textit{then}\;\textit{ALT}\;\textit{bs}\;\;(\textit{SEQ}\;[]\;(r_1\backslash c)\;r_2)\;\;(\textit{fuse}\;(\textit{bmkeps}\;r_1)\;(r_2\backslash c))\\
& & \textit{else}\;\textit{SEQ}\;\textit{bs}\;(r_1\backslash c)\;r_2\\
(\textit{STAR}\;\textit{bs}\;r)\backslash c & \stackrel{\text{def}}{=} & \textit{SEQ}\;\;(\textit{bs}\;@\;[Z])\;(r\backslash c)\;(\textit{STAR}\;[]\;r)\\
(\textit{NT}\;\textit{bs}\;r\;n)\backslash c & \stackrel{\text{def}}{=} & \textit{if}\;n = 0\;\textit{then}\;\textit{ZERO}\;\textit{else}\;\textit{SEQ}\;\;(\textit{bs}\;@\;[Z])\;(r\backslash c)\;(\textit{NT}\;[]\;r\;(n-1))
\end{array}
$$

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*:

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

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

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

$$\begin{array}{lcl}
\textit{retrieve}\ (\textit{ONE}\ \textit{bs})\ (\textit{Empty}) & \stackrel{\text{def}}{=} & \textit{bs}\\
\textit{retrieve}\ (\textit{CHAR}\ \textit{bs}\ c)\ (\textit{Char}\ d) & \stackrel{\text{def}}{=} & \textit{bs}\\
\textit{retrieve}\ (\textit{ALTs}\ \textit{bs}\ [r])\ v & \stackrel{\text{def}}{=} & \textit{bs}\ @\ \textit{retrieve}\ r\ v\\
\textit{retrieve}\ (\textit{ALTs}\ \textit{bs}\ (r :: \textit{rs}))\ (\textit{Left}\ v) & \stackrel{\text{def}}{=} & \textit{bs}\ @\ \textit{retrieve}\ r\ v\\
\textit{retrieve}\ (\textit{ALTs}\ \textit{bs}\ (r :: \textit{rs}))\ (\textit{Right}\ v) & \stackrel{\text{def}}{=} & \textit{bs}\ @\ \textit{retrieve}\ (\textit{ALTs}\ []\ \textit{rs})\ v\\
\textit{retrieve}\ (\textit{SEQ}\ \textit{bs}\ r_1\ r_2)\ (\textit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} & \textit{bs}\ @\ \textit{retrieve}\ r_1\ v_1\ @\ \textit{retrieve}\ r_2\ v_2\\
\textit{retrieve}\ (\textit{STAR}\ \textit{bs}\ r)\ (\textit{Stars}\ []) & \stackrel{\text{def}}{=} & \textit{bs}\ @\ [S]\\
\textit{retrieve}\ (\textit{STAR}\ \textit{bs}\ r)\ (\textit{Stars}\ (v :: \textit{vs})) & \stackrel{\text{def}}{=} & \textit{bs}\ @\ [Z]\ @\ \textit{retrieve}\ r\ v\ @\ \textit{retrieve}\ (\textit{STAR}\ []\ r)\ (\textit{Stars}\ \textit{vs})\\
\textit{retrieve}\ (\textit{NT}\ \textit{bs}\ r\ 0)\ (\textit{Stars}\ []) & \stackrel{\text{def}}{=} & \textit{bs}\ @\ [S]\\
\textit{retrieve}\ (\textit{NT}\ \textit{bs}\ r\ (n+1))\ (\textit{Stars}\ (v :: \textit{vs})) & \stackrel{\text{def}}{=} & \textit{bs}\ @\ [Z]\ @\ \textit{retrieve}\ r\ v\ @\ \textit{retrieve}\ (\textit{NT}\ []\ r\ n)\ (\textit{Stars}\ \textit{vs})
\end{array}$$

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 $\textit{Left}$ then we descend into the left-hand side of an alternative in order to assemble the bitcode. Similarly for $\textit{Right}$. 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 5.** *If* $\vdash v : r$ *then* $\textit{code}\,v = \textit{retrieve}\,(r^{\uparrow})\,v$.

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 6.** *(1)* $(r\backslash s)^{\downarrow} = (r^{\downarrow})\backslash s$
(2) $\textit{bnullable}(r)$ *iff* $\textit{nullable}(r^{\downarrow})$
(3) $\textit{bmkeps}(r) = \textit{retrieve}\,r\,(\textit{mkeps}\,(r^{\downarrow}))$ *provided* $\textit{nullable}(r^{\downarrow})$

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* (which has two phases) by stacking up injection functions during the forward phase. An auxiliary function, called *flex*, allows us to recast the rules of $\textit{lexer}$ in terms of a single phase and stacked up injection functions.

$$\textit{flex}\ r\ f\ [] \stackrel{\text{def}}{=} f \qquad\qquad \textit{flex}\ r\ f\ (c{::}s) \stackrel{\text{def}}{=} \textit{flex}\,(r\backslash c)\,(\lambda v.\ f\,(\textit{inj}\,r\,c\,v))\,s$$

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

▶ **Lemma 7.** *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), it helps us with proving that incrementally building up values in *blexer* 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 *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 8** (Main Lemma). *If* $\vdash v : r\backslash s$ *then* $\textit{Some}\,(\textit{flex}\,r\,\textit{id}\,s\,v) = \textit{decode}(\textit{retrieve}\,(r^{\uparrow}\backslash s)\,v)\,r$

We can then prove the correctness of *blexer*—it indeed produces the same result as *lexer*.

▶ **Theorem 9.** $\textit{lexer}\,r\,s = \textit{blexer}\,r\,s$

This establishes that the bitcoded algorithm *without* simplification produces correct results. This was only conjectured by Sulzmann and Lu in their paper [14]. The next step is to add simplifications.

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

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

This is a simpler derivative, but unfortunately we cannot make any further simplifications. This is a problem because the outermost alternatives contains two copies of the same regular expression (underlined with r). These copies will spawn new copies in later derivative steps and they in turn even more copies. This destroys any 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 it could be safely removed without affecting the correctness of the algorithm. 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 not next to each other but separated by another 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 with the bitsequences in bitcoded regular expressions is that they can be easily modified such that simplification does not interfere with the value constructions. For example we can "flatten", or de-nest, or spill out, *ALTs* as follows

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

where we just need to fuse the bitsequence that has accumulated in $\textit{bs}_2$ to the alternatives in $\textit{rs}_2$. 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 simplification rules described by Sulzmann and Lu. Recasting their definition with our syntax they define the step of removing duplicates as

$$\mathit{bsimp}\ (\mathit{ALTs}\ \mathit{bs}\ \mathit{rs}) \overset{\text{def}}{=} \mathit{ALTs}\ \mathit{bs}\ (\mathit{nub}\ (\mathit{map}\ \mathit{bsimp}\ \mathit{rs}))$$

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 elements in *rs* will have a different annotated bitsequence and in this way *nub* will never find a duplicate to be removed. One 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 *distinctWith rs eq acc* where *eq* is an user-defined equivalence relation that compares two elements in *rs*. We define this function in Isabelle/HOL as

$$\begin{array}{lcl}
\mathit{distinctWith}\ []\ \mathit{eq}\ \mathit{acc} & \overset{\text{def}}{=} & [] \\
\mathit{distinctWith}\ (x :: \mathit{xs})\ \mathit{eq}\ \mathit{acc} & \overset{\text{def}}{=} & \mathit{if}\ (\exists\ y \in \mathit{acc}.\ \mathit{eq}\ x\ y)\ \mathit{then}\ \mathit{distinctWith}\ \mathit{xs}\ \mathit{eq}\ \mathit{acc} \\
 & & \qquad\qquad\qquad\qquad\quad \mathit{else}\ x :: \mathit{distinctWith}\ \mathit{xs}\ \mathit{eq}\ (\{x\} \cup \mathit{acc})
\end{array}$$

where we scan the list from left to right (because we have to remove later copies). In *distinctWith*, *eq* is intended to be an equivalence relation for annotated regular expressions and *acc* is an accumulator for annotated regular expressions—essentially a set of regular expressions that we have already seen while scanning the list. Therefore we delete an element, say *x*, from the list provided a *y* with *y* being equivalent to *x* is already in the accumulator; otherwise we keep *x* and scan the rest of the list but add *x* as another "seen" element to *acc*. We will use *distinctWith* where *eq* is an equivalence that deletes bitsequences from bitcoded regular expressions before comparing the components. One way to define this in Isabelle/HOL is by the following recursive function from annotated regular expressions to *bool*:

$$\begin{array}{lcl}
\mathit{ZERO} \approx \mathit{ZERO} & \overset{\text{def}}{=} & \mathit{True} \\
\mathit{ONE}\ \_ \approx \mathit{ONE}\ \_ & \overset{\text{def}}{=} & \mathit{True} \\
\mathit{CHAR}\ \_\ c \approx \mathit{CHAR}\ \_\ d & \overset{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \mathit{True}\ \mathit{else}\ \mathit{False} \\
\mathit{SEQ}\ \_\ r_{11}\ r_{12} \approx \mathit{SEQ}\ \_\ r_{21}\ r_{22} & \overset{\text{def}}{=} & r_{11} \approx r_{21} \wedge r_{12} \approx r_{22} \\
\mathit{ALTs}\ \_\ [] \approx \mathit{ALTs}\ \_\ [] & \overset{\text{def}}{=} & \mathit{True} \\
\mathit{ALTs}\ \_\ (r_1 :: \mathit{rs}_1) \approx \mathit{ALTs}\ \_\ (r_2 :: \mathit{rs}_2) & \overset{\text{def}}{=} & r_1 \approx r_2 \wedge \mathit{ALTs}\ \_\ \mathit{rs}_1 \approx \mathit{ALTs}\ \_\ \mathit{rs}_2 \\
\mathit{STAR}\ \_\ r_1 \approx \mathit{STAR}\ \_\ r_2 & \overset{\text{def}}{=} & r_1 \approx r_2 \\
\mathit{NT}\ \_\ r_1\ n_1 \approx \mathit{NT}\ \_\ r_2\ n_2 & \overset{\text{def}}{=} & r_1 \approx r_2 \wedge n_1 = n_2
\end{array}$$

where all other cases are set to *False*. This equivalence is clearly a computationally more expensive operation than *nub*, but is needed in order to make the removal of unnecessary copies to work properly.

Our simplification function depends on three more helper functions, one is called *flts* and analyses lists of regular expressions coming from alternatives. It is defined as follows:

$$\mathit{flts}\ [] \overset{\text{def}}{=} [] \qquad\qquad \mathit{flts}\ (\mathit{ZERO} :: \mathit{rs}) \overset{\text{def}}{=} \mathit{flts}\ \mathit{rs}$$
$$\mathit{flts}\ (\mathit{ALTs}\ \mathit{bs}'\ \mathit{rs}' :: \mathit{rs}) \overset{\text{def}}{=} \mathit{map}\ (\mathit{fuse}\ \mathit{bs}')\ \mathit{rs}'\ @\ \mathit{flts}\ \mathit{rs}$$

The second clause of *flts* removes all instances of *ZERO* in alternatives and the third "de-nests" 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:

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

With this in place we can define our simplification function as

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

We believe our recursive function *bsimp* simplifies bitcoded regular expressions as intended by Sulzmann and Lu. There is no point in applying the *bsimp* function repeatedly (like the simplification in their paper which needs to be applied until a fixpoint is reached) because we can show that *bsimp* is idempotent, that is

▶ **Proposition 10.** $\mathit{bsimp}\ (\mathit{bsimp}\ r) = \mathit{bsimp}\ r$

This can be proved by induction on r but requires a detailed analysis that the de-nesting of alternatives always results in a flat list of regular expressions. We omit the details since it does not concern the correctness proof.

Next we can include simplification after each derivative step leading to the following notion of bitcoded derivatives:

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

and use it in the improved lexing algorithm defined as

$$\mathit{blexer}^{+}\ r\ s\ \stackrel{\text{def}}{=}\ \mathit{let}\ r_{der} = (r^{\uparrow})\backslash_{bsimp}\ s\ \mathit{in}\quad \mathit{if}\ \mathit{bnullable}(r_{der})\quad \mathit{then}\ \mathit{decode}\ (\mathit{bmkeps}\ r_{der})\ r\quad \mathit{else}\ \mathit{None}$$

The remaining task is to show that *blexer* and blexer^{+} generate the same answers.

When we first attempted this proof we encountered a problem with the idea in Sulzmann and Lu's paper where the argument seems to be to appeal again to the *retrieve*-function defined for the unsimplified version of the algorithm. But this does not work, because desirable properties such as

$$\mathit{retrieve}\ r\ v = \mathit{retrieve}\ (\mathit{bsimp}\ r)\ v$$

do not hold under simplification—this property essentially purports that we can retrieve the same value from a simplified version of the regular expression. To start with *retrieve* depends on the fact that the value v corresponds to the structure of the regular expression r—but the whole point of simplification is to "destroy" this structure by making the regular expression simpler. To see this consider the regular expression $r = r' + \mathbf{0}$ and a corresponding value $v = \mathit{Left}\ v'$. If we annotate bitcodes to r, then we can use *retrieve* with r and v in order to extract a corresponding bitsequence. The reason that this works is that r is an alternative regular expression and v a corresponding *Left*-value. However, if we simplify r, then v does not correspond to the shape of the regular expression anymore. So unless one can somehow synchronise the change in the simplified regular expressions with the original POSIX value, there is no hope of appealing to *retrieve* in the correctness argument for blexer^{+}.

We found it more helpful to introduce the rewriting systems shown in Figure 3. The idea is to generate simplified regular expressions in small steps (unlike the *bsimp*-function which does the same in a big step), and show that each of the small steps preserves the bitcodes that lead to the final POSIX value. The rewrite system is organised such that $\rightsquigarrow$ is for bitcoded regular expressions and $\overset{s}{\rightsquigarrow}$ for lists of bitcoded regular expressions. The former essentially implements the simplifications of *bsimpSEQ* and *flts*; while the latter implements the simplifications in *bsimpALTs*. We can show that any bitcoded regular expression reduces in zero or more steps to the simplified regular expression generated by *bsimp*:

$$\frac{}{(SEQ\ bs\ ZERO\ r_2) \rightsquigarrow (ZERO)}\,S\mathbf{0}_l \qquad \frac{}{(SEQ\ bs\ r_1\ ZERO) \rightsquigarrow (ZERO)}\,S\mathbf{0}_r \qquad \frac{}{(SEQ\ bs_1\ (ONE\ bs_2)\ r) \rightsquigarrow fuse\ (bs_1\ @\ bs_2)\ r}\,S\mathbf{1}$$

$$\frac{r_1 \rightsquigarrow r_2}{(SEQ\ bs\ r_1\ r_3) \rightsquigarrow (SEQ\ bs\ r_2\ r_3)}\,\mathrm{SL} \qquad \frac{r_3 \rightsquigarrow r_4}{(SEQ\ bs\ r_1\ r_3) \rightsquigarrow (SEQ\ bs\ r_1\ r_4)}\,\mathrm{SR}$$

$$\frac{}{(ALTs\ bs\ []) \rightsquigarrow (ZERO)}\,A0 \qquad \frac{}{(ALTs\ bs\ [r]) \rightsquigarrow fuse\ bs\ r}\,A1 \qquad \frac{rs_1 \overset{s}{\rightsquigarrow} rs_2}{(ALTs\ bs\ rs_1) \rightsquigarrow (ALTs\ bs\ rs_2)}\,AL$$

$$\frac{rs_1 \overset{s}{\rightsquigarrow} rs_2}{r :: rs_1 \overset{s}{\rightsquigarrow} r :: rs_2}\,LT \qquad \frac{r_1 \rightsquigarrow r_2}{r_1 :: rs \overset{s}{\rightsquigarrow} r_2 :: rs}\,LH \qquad \frac{}{ZERO :: rs \overset{s}{\rightsquigarrow} rs}\,L\mathbf{0} \qquad \frac{}{ALTs\ bs\ rs_1 :: rs_2 \overset{s}{\rightsquigarrow} (map\ (fuse\ bs)\ rs_1\ @\ rs_2)}\,LS$$

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

Figure 3 The rewrite rules that generate simplified regular expressions in small steps: $r_1 \rightsquigarrow r_2$ is for bitcoded regular expressions and $rs_1 \overset{s}{\rightsquigarrow} rs_2$ for *lists* of bitcoded expressions. Interesting is the LD rule that allows copies of regular expressions to be removed provided a regular expression earlier in the list can match the same strings.

▶ **Lemma 11.** $r \rightsquigarrow^* bsimp\ r$

We can also show that this rewrite system preserves *bnullable*, that is simplification does not affect nullability:

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

From this, we can show that *bmkeps* will produce the same bitsequence as long as one of the bitcoded regular expressions in $\rightsquigarrow$ is nullable (this lemma establishes the missing fact we were not able to establish using *retrieve*, as suggested in the paper by Sulzmann and Lu).

▶ **Lemma 13.** *If* $r_1 \rightsquigarrow r_2$ *and* $bnullable\ r_1 \wedge bnullable\ r_2$ *then* $bmkeps\ r_1 = bmkeps\ r_2$.

Crucial is also the fact that derivative steps and simplification steps can be interleaved, which is shown by the fact that $\rightsquigarrow$ is preserved under derivatives.

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

Using this fact together with Lemma 11 allows us to prove the central lemma that the unsimplified derivative (with a string s) reduces to the simplified derivative (with the same string).

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

With these lemmas in place we can finally establish that *blexer*$^+$ and *blexer* generate the same value, and using Theorem 9 from the previous section that this value is indeed the POSIX value.

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

This means that if the algorithm is called with a regular expression r and a string s with $s \in L(r)$, it will return *Some* v for the unique v we defined by the POSIX relation $(s, r) \to v$; otherwise the algorithm returns *None* when $s \notin L(r)$ and no such v exists. This completes the correctness proof for the second POSIX lexing algorithm by Sulzmann and Lu. The interesting point of this algorithm is that the sizes of derivatives do not grow arbitrarily big but can be finitely bounded, which we shall show next.

5 Finiteness of Derivatives

In this section let us sketch our argument for why the size of the simplified derivatives with the aggressive simplification function can be finitely bounded. Suppose we have a size function for bitcoded regular expressions, written $[\![r]\!]$, which counts the number of nodes if we regard r as a tree (we omit the precise definition; ditto for lists $[\![rs]\!]$). For this we show that for every r there exists a bound N such that

$$\forall s.\ [\![r\backslash_{bsimp}\, s]\!] \leq N$$

Note that the bound N is a bound for *all* strings, no matter how long they are. We establish this bound by induction on r. The base cases for *ZERO*, *ONE bs* and *CHAR bs c* are straightforward. The interesting case is for sequences of the form *SEQ bs* r_1 r_2. In this case our induction hypotheses state $\exists N_1.\forall s.\ [\![r_1\backslash_{bsimp}\, s]\!] \leq N_1$ and $\exists N_2.\forall s.\ [\![r_2\backslash_{bsimp}\, s]\!] \leq N_2$. We can reason as follows

$$\begin{array}{cll}
 & [\![(\mathit{SEQ}\ \mathit{bs}\ r_1\ r_2)\backslash_{bsimp}\, s]\!] & \\
= & [\![\mathit{bsimp}\,(\mathit{ALTs}\ \mathit{bs}\ ((\mathit{SEQ}\ [\,]\ (r_1\backslash_{bsimp}\, s)\ r_2) :: [r_2\backslash_{bsimp}\, s' \mid s' \in \mathit{Suffix}(r_1, s)]))]\!] & (1)\\
\leq & [\![\mathit{distinctWith}\,((\mathit{SEQ}\ [\,]\ (r_1\backslash_{bsimp}\, s)\ r_2) :: [r_2\backslash_{bsimp}\, s' \mid s' \in \mathit{Suffix}(r_1, s)])\ \approx\ \varnothing]\!] + 1 & (2)\\
\leq & [\![\mathit{SEQ}\ [\,]\ (r_1\backslash_{bsimp}\, s)\ r_2]\!] + [\![\mathit{distinctWith}\,[r_2\backslash_{bsimp}\, s' \mid s' \in \mathit{Suffix}(r_1, s)]\ \approx\ \varnothing]\!] + 1 & (3)\\
\leq & N_1 + [\![r_2]\!] + 2 + [\![\mathit{distinctWith}\,[r_2\backslash_{bsimp}\, s' \mid s' \in \mathit{Suffix}(r_1, s)]\ \approx\ \varnothing]\!] & (4)\\
\leq & N_1 + [\![r_2]\!] + 2 + l_{N_2} * N_2 & (5)
\end{array}$$

where in (1) the $\mathit{Suffix}(r_1, s)$ are all the suffixes of s where $r_1\backslash_{bsimp}\, s'$ is nullable (s' being a suffix of s). In (3) we know that $[\![\mathit{SEQ}\ [\,]\ (r_1\backslash_{bsimp}\, s)\ r_2]\!]$ is bounded by $N_1 + [\![r_2]\!] + 1$. In (5) we know the list comprehension contains only regular expressions of size smaller than N_2. The list length after *distinctWith* is bounded by a number, which we call l_{N_2}. It stands for the number of distinct regular expressions smaller than N_2 (there can only be finitely many of them). We reason similarly for *STAR*.

Clearly we give in this finiteness argument (Step (5)) a very loose bound that is far from the actual bound we can expect. We can do better than this, but this does not improve the finiteness property we are proving. If we are interested in a polynomial bound, one would hope to obtain a similar tight bound as for partial derivatives introduced by Antimirov [2]. After all the idea with *distinctWith* is to maintain a “set” of alternatives (like the sets in partial derivatives). Unfortunately to obtain the exact same bound would mean we need to introduce simplifications, such as $(r_1 + r_2) \cdot r_3 \longrightarrow (r_1 \cdot r_3) + (r_2 \cdot r_3)$, which exist for partial derivatives. However, if we introduce them in our setting we would lose the POSIX property of our calculated values. For example given the regular expressions $(a + ab) \cdot (b + \mathbf{1})$ and the string $[a, b]$, then our algorithm generates the following correct POSIX value

$$\mathit{Seq}\ (\mathit{Right}\ (\mathit{Seq}\ (\mathit{Char}\ a)\ (\mathit{Char}\ b)))\ (\mathit{Right}\ \mathit{Empty})$$

Essentially it matches the string with the longer *Right*-alternative in the first sequence (and then the ‘rest’ with the empty regular expression **1** from the second sequence). If we add the simplification above, then we obtain the following value

$$\mathit{Seq}\ (\mathit{Left}\ (\mathit{Char}\ a))\ (\mathit{Left}\ (\mathit{Char}\ b))$$

where the *Left*-alternatives get priority. However, this violates the POSIX rules and we have not been able to reconcile this problem. Therefore we leave better bounds for future work.

6 Conclusion

We set out in this work to prove in Isabelle/HOL the correctness of the second POSIX lexing algorithm by Sulzmann and Lu [14]. This follows earlier work where we established the correctness of the first algorithm [3]. In the earlier work we needed to introduce our own specification about what POSIX values are, because the informal definition given by Sulzmann and Lu did not stand up to a formal proof. Also for the second algorithm we needed to introduce our own definitions and proof ideas in order to establish the correctness. Our interest in the second algorithm lies in the fact that by using bitcoded regular expressions and an aggressive simplification method there is a chance that the derivatives can be kept universally small (we established in this paper that they can be kept finitely bounded for any string).

Having proved the correctness of the POSIX lexing algorithm, which lessons have we learned? Well, we feel this is a very good example where formal proofs give further insight into the matter at hand. For example it is very hard to see a problem with *nub* vs *distinctWith* with only experimental data—one would still see the correct result but find that simplification does not simplify in well-chosen, but not obscure, examples.

We can of course only make a claim about the correctness and the sizes of the derivatives, not about the efficiency or runtime of our version of Sulzman and Lu's algorithm. But we found the size is an important first indicator about efficiency: clearly if the derivatives can grow to arbitrarily big sizes and the algorithm needs to traverse the derivatives possibly several times, then the algorithm will be slow—excruciatingly slow that is. Other works seems to make stronger claims, but during our work we have developed a healthy suspicion when for example experimental data is used to back up efficiency claims. For example Sulzmann and Lu write about their equivalent of $blexer^+$ *"...we can incrementally compute bitcoded parse trees in linear time in the size of the input"* [14, Page 14]. Given the growth of the derivatives in some cases even after aggressive simplification, this is a hard to believe claim. A similar claim about a theoretical runtime of $O(n^2)$ is made for the Verbatim lexer, which calculates tokens according to POSIX rules [7]. For this, Verbatim uses Brzozowski's derivatives like in our work. The authors write: *"The results of our empirical tests [..] confirm that Verbatim has* $O(n^2)$ *time complexity."* [7, Section VII]. While their correctness proof for Verbatim is formalised in Coq, the claim about the runtime complexity is only supported by some emperical evidence obtained by using the code extraction facilities of Coq. Given our observation with the "growth problem" of derivatives, we tried out their extracted OCaml code with the example $(a + aa)^*$ as a single lexing rule, and it took for us around 5 minutes to tokenise a string of 40 a's and that increased to approximately 19 minutes when the string is 50 a's long. Taking into account that derivatives are not simplified in the Verbatim lexer, such numbers are not surprising. Clearly our result of having finite derivatives might sound rather weak in this context but we think such effeciency claims really require further scrutiny.

The contribution of this paper is to make sure derivatives do not grow arbitrarily big (universially) In the example $(a + aa)^*$, *all* derivatives have a size of 17 or less. The result is that lexing a string of, say, 50 000 a's with the regular expression $(a + aa)^*$ takes approximately 10 seconds with our Scala implementation of the presented algorithm.

Finally, let us come back to the point about bounded regular expressions. We have in this paper only shown that $r^{\{n\}}$ can be included, but all our results extend straightforwardly also to the other bounded regular expressions. We find bounded regular expressions fit naturally into the setting of Brzozowski derivatives and the bitcoded regular expressions by Sulzmann and Lu. In contrast bounded regular expressions are often the Achilles' heel in regular expression matchers that use the traditional automata-based approach to lexing, primarily because they need to expand the counters of bounded regular expressions into n-connected copies of an automaton. This is not needed in Sulzmann and Lu's algorithm. To see the difference consider for example the regular expression $a^{\{1001\}} \cdot a^*$, which is not permitted in the Go language because the counter is too big. In contrast we have no problem with matching this regular expression with, say 50 000 a's, because the counters can be kept compact. In fact, the overall size of the derivatives is never greater than 5 in this example. Even in the example from Section 2, where Rust raises an error message, namely $a^{\{1000\}\{100\}\{5\}}$, the maximum size for our derivatives is a moderate 14.

Let us also compare our work to the verified Verbatim++ lexer where the authors of the Verbatim lexer introduced a number of improvements and optimisations, for example memoisation [8]. However, unlike Verbatim, which works with derivatives like in our work, Verbatim++ compiles first a regular expression into a DFA. While this makes lexing fast in many cases, with examples of bounded regular expressions like $a^{\{100\}\{5\}}$ one needs to represent them as sequences of $a \cdot a \cdot \ldots \cdot a$ (500 a's in sequence). We have run their extracted code with such a regular expression as a single lexing rule and a string of 50 000 a's—lexing in this case takes approximately 5 minutes. We are not aware of any better translation using the traditional notion of DFAs. Therefore we prefer to stick with calculating derivatives, but attempt to make this calculation (in the future) as fast as possible. What we can guaranty with the presented work is that the maximum size of the derivatives for this example is not bigger than 9. This means our Scala implementation only needs a few seconds for this example.

Our Isabelle code including the results from Sec. 5 is available from https://github.com/urbanchr/posix.

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 V. Antimirov. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science*, 155:291–319, 1995.

3 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.

4 H. Björklund, W. Martens, and T. Timm. Efficient Incremental Evaluation of Succinct Regular Expressions. In *Proc. of the 24th ACM Conf. on Information and Knowledge Management (CIKM)*, pages 1541–1550, 2015.

5 J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

6 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.

7 D. Egolf, S. Lasser, and K. Fisher. Verbatim: A Verified Lexer Generator. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 92–100, 2021.

8 D. Egolf, S. Lasser, and K. Fisher. Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Dderivatives. In *Proc. of the 11th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*, pages 27–39. ACM, 2022.

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

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

11 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.

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

13 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.

14 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.

15 L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar. Regex Matching with Counting-Set Automata. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4:218:1–218:30, 2020.

16 S. Vansummeren. Type Inference for Unique Pattern Matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.

A Some Proofs

While we have formalised *all* results in Isabelle/HOL, below we shall give some rough details of our reasoning in “English” if this is helpful for reviewing.

Proof of Lemma 4. 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 $\vdash v : r$. ◀

Proof of Lemma 8. 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 4 and 5. In the case $s\,@\,[c]$, we can infer from the assumption that $\vdash v : (r\backslash s)\backslash c$ holds. Hence by Prop. 2 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 IH 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. ◀

Proof of Theorem 9. We can first expand both sides using Lem. 7 and the definition of *blexer*. This gives us two *if*-statements, which we need to show to be equal. By Lemma 6*(2)* we know the *if*-tests coincide: $\mathit{bnullable}(r^{\uparrow}\backslash s)$ *iff* $\mathit{nullable}(r\backslash s)$. For the *if*-branch suppose $r_d \overset{\text{def}}{=} r^{\uparrow}\backslash s$ and $d \overset{\text{def}}{=} r\backslash s$. We have (*) $\mathit{nullable}\;d$. We can then show by Lemma 6*(3)* that

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

where the right-hand side is equal to $\mathit{Some}\,(\mathit{flex}\;r\;\mathit{id}\;s\;(\mathit{mkeps}\;d))$ by Lemma 8 (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. ◀

Proof of Lemma 11. By induction on r. For this we can use the properties $rs \overset{s}{\leadsto}{}^{*}\, \mathit{flts}\;rs$ and $rs \overset{s}{\leadsto}{}^{*}\, \mathit{distinctWith}\;rs \approx \varnothing$. The latter uses repeated applications of the LD rule which allows the removal of duplicates that can recognise the same strings. ◀

Proof of Lemma 12. Straightforward mutual induction on the definition of $\leadsto$ and $\overset{s}{\leadsto}$. The only interesting case is the rule LD where the property holds since by the side-conditions of that rule the empty string will be in both $L\;(rs_a\;@\;[r_1]\;@\;rs_b\;@\;[r_2]\;@\;rs_c)$ and $L\;(rs_a\;@\;[r_1]\;@\;rs_b\;@\;rs_c)$. ◀

Proof of Lemma 13. By straightforward mutual induction on the definition of $\leadsto$ and $\overset{s}{\leadsto}$. Again the only interesting case is the rule LD where we need to ensure that $\mathit{bmkeps}\;(rs_a\;@\;[r_1]\;@\;rs_b\;@\;[r_2]\;@\;rs_c) = \mathit{bmkeps}\;(rs_a\;@\;[r_1]\;@\;rs_b\;@\;rs_c)$ holds. This is indeed the case because according to the POSIX rules the generated bitsequence is determined by the first alternative that can match the string (in this case being nullable). ◀

Proof of Lemma 14. By straightforward mutual induction on the definition of $\leadsto$ and $\overset{s}{\leadsto}$. The case for LD holds because $L\;((r_2\backslash c)^{\downarrow}) \subseteq L\;((r_1\backslash c)^{\downarrow})$ if and only if $L\;(r_2{}^{\downarrow}) \subseteq L\;(r_1{}^{\downarrow})$. ◀

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