POSIX Lexing with Derivatives of Regular Expressions

Christian Urban

King's College London, UK.

Corresponding author(s). E-mail(s): christian.urban@kcl.ac.uk;

Abstract

Brzozowski introduced the notion of derivatives for regular expressions. They can be used for a very simple regular expression matching algorithm. Sulzmann and Lu cleverly extended this algorithm in order to deal with POSIX matching, which is the underlying disambiguation strategy for regular expressions needed in lexers. 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. In this paper we give our inductive definition of what a POSIX value is and show that Sulzmann and Lu's algorithm always generates such a value. We also show that our inductive definition of a POSIX value is equivalent to an alternative definition by Okui and Suzuki which identifies POSIX values as least elements according to an ordering of values.

1 Introduction

Brzozowski [5] introduced the notion of the *derivative* $r\backslash c$ of a regular expression r w.r.t. a character c, and showed that it gave 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

*This paper is a revised and expanded version of [1]. Compared with that paper we give a second definition for POSIX values introduced by Okui and Suzuki [2, 3] and prove that it is equivalent to our original one. This second definition is based on an ordering of values and very similar to, but not equivalent with, the definition given by Sulzmann and Lu [4]. The advantage of the definition based on the ordering is that it implements more directly the informal rules from the POSIX standard. Furthermore we extend our results to bounded repetitions of regular expressions, records and character sets.

string, then r matches s (and *vice versa*). The derivative has the property (which may almost be regarded as its specification) that, for every string s and regular expression r and character c, one has $cs \in L(r)$ if and only if $s \in L(r\backslash c)$. The beauty of Brzozowski's derivatives is that they are neatly expressible in any functional programming language, and easily definable and reasoned about in theorem provers—the definitions just consist of inductive datatypes and simple recursive functions. A mechanised correctness proof of Brzozowski's matcher in for example HOL4 has been mentioned by Owens and Slind [6]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [7]. And another one in Coq is given by Coquand and Siles [8]. Also Ribeiro and Du Bois give one in Agda [9].

If a regular expression matches a string, then in general there is more than one way of how the string is matched. There are two commonly used disambiguation strategies to generate a unique answer: one is called GREEDY matching [10] and the other is POSIX matching [2, 4, 11–13].[1] For example consider the string xy and the regular expression $(x + y + xy)^*$. Either the string can be matched in two 'iterations' by the single letter-regular expressions x and y, or directly in one iteration by xy. The first case corresponds to GREEDY matching, which first matches with the left-most symbol and only matches the next symbol in case of a mismatch (this is greedy in the sense of preferring instant gratification to delayed repletion). The second case is POSIX matching, which prefers the longest match.

In the context of lexing, where an input string needs to be split up into a sequence of tokens, POSIX is the more natural disambiguation strategy for what programmers consider basic syntactic building blocks in their programs. These building blocks are often specified by some regular expressions, say r_{key} and r_{id} for recognising keywords and identifiers, respectively. There are a few underlying (informal) rules behind tokenising a string in a POSIX [11] fashion:

- *The Longest Match Rule* (or *"Maximal Munch Rule"*): The longest initial substring matched by any regular expression is taken as next token.
- *Priority Rule:* For a particular longest initial substring, the first (leftmost) regular expression that can match determines the token.
- *Star Rule:* A subexpression repeated by * shall not match an empty string unless this is the only match for the repetition.
- *Empty String Rule:* An empty string shall be considered to be longer than no match at all.

Consider for example the regular expression r_{key} for recognising keywords such as *if*, *then*, *while* and so on; and r_{id} for recognising identifiers (say, a single character followed by characters or numbers). Then we can form the regular expression $(r_{key} + r_{id})^*$ and use POSIX matching to tokenise strings, say *iffoo* and *if*. For *iffoo* we obtain by the Longest Match Rule a single identifier token, not a keyword followed by an identifier. For *if* we obtain by the Priority Rule

[1]POSIX matching acquired its name from the fact that the corresponding rules were described as part of the POSIX specification for Unix-like operating systems [11].

a keyword token, not an identifier token—even if r_{id} matches also. By the Star Rule we know $(r_{key} + r_{id})^*$ matches *iffoo*, respectively *if*, in exactly one 'iteration' of the star. The Empty String Rule is for cases where, for example, the regular expression $(a^*)^*$ matches against the string *bc*. Then the longest initial matched substring is the empty string, which is matched by both the whole regular expression and the parenthesised subexpression.

One limitation of Brzozowski's matcher is that it only generates a YES/NO answer for whether a string is being matched by a regular expression. Sulzmann and Lu [4] extended this matcher to allow generation not just of a YES/NO answer but of an actual matching, called a *lexical value*. Assuming a regular expression matches a string, values encode the information of *how* the string is matched by the regular expression—that is, which part of the string is matched by which part of the regular expression. For this consider again the string xy and the regular expression $(x + (y + xy))^*$ (this time fully parenthesised). We can view this regular expression as a tree and if the string xy is matched by two Star 'iterations', then the x is matched by the left-most alternative in this tree and the y by the right-left alternative. This suggests to record this matching as

$$\mathit{Stars}\ [\mathit{Left}\ (\mathit{Char}\ x),\ \mathit{Right}\ (\mathit{Left}\ (\mathit{Char}\ y))]$$

where *Stars*, *Left*, *Right* and *Char* are constructors for values. *Stars* records how many iterations were used; *Left*, respectively *Right*, which alternative is used. The value for matching xy in a single 'iteration', i.e. the POSIX value, would look as follows

$$\mathit{Stars}\ [\mathit{Right}\ (\mathit{Right}\ (\mathit{Seq}\ (\mathit{Char}\ x)\ (\mathit{Char}\ y)))]$$

where *Stars* has only a single-element list for the single iteration and *Seq* indicates that xy is matched by a sequence regular expression. This 'tree view' leads naturally to the idea that regular expressions act as types and values as inhabiting those types (see, for example, [14, 15]).

Sulzmann and Lu give a simple algorithm to calculate a value that appears to be the value associated with POSIX matching. The challenge then is to specify that value, in an algorithm-independent fashion, and to show that Sulzmann and Lu's derivative-based algorithm does indeed calculate a value that is correct according to the specification. The answer given by Sulzmann and Lu [4] is to define a relation (called an "order relation") on the set of values of r, and to show that (once a string to be matched is chosen) there is a maximum element and that it is computed by their derivative-based algorithm. This proof idea is inspired by work of Frisch and Cardelli [10] on a GREEDY regular expression matching algorithm. However, we were not able to establish transitivity and totality for the "order relation" by Sulzmann and Lu. There are some inherent problems with their approach (of which some of the proofs are not published in [4]); perhaps more importantly, we give in this paper a simple inductive (and algorithm-independent) definition of what we call being a *POSIX value* for a regular expression r and a string s; we show that the algorithm by Sulzmann and Lu computes such a value and that such a value is unique. Our proofs are both done by hand and checked in Isabelle/HOL. The experience of doing our proofs has been that this mechanical checking was absolutely essential: this subject area has hidden snares. This was also noted by Kuklewicz [12] who found that nearly all POSIX matching implementations are "buggy" [4, Page 203] and by Grathwohl et al [16, Page 36] who wrote:

> *"The POSIX strategy is more complicated than the greedy because of the dependence on information about the length of matched strings in the various subexpressions."*

Contributions: We have implemented in Isabelle/HOL the derivative-based regular expression matching algorithm of Sulzmann and Lu [4]. We have proved the correctness of this algorithm according to our specification of what a POSIX value is (inspired by work of Vansummeren [13]). Sulzmann and Lu sketch in [4] an informal correctness proof: but to us it contains unfillable gaps.[2] Our specification of a POSIX value consists of a simple inductive definition that given a string and a regular expression uniquely determines this value. We also show that our definition is equivalent to an ordering of values based on positions by Okui and Suzuki [2].

2 Preliminaries

Strings in Isabelle/HOL are lists of characters with the empty string being represented by the empty list, written $[]$, and list-cons being written as $\_ :: \_$. Often we use the usual bracket notation for lists also for strings; for example a string consisting of just a single character c is written $[c]$; the string abc is written $[a,\ b,\ c]$. We use the usual definitions for *prefixes* and *strict prefixes* of strings. By using the type *char* for characters we have a supply of finitely many characters roughly corresponding to the ASCII character set. Regular expressions are defined as usual as the elements of the following inductive datatype:

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

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. We use $+$ and $\cdot$ for alternative and sequence regular expressions, respectively. We are adding here to the usual regular expressions also the regular expression for character sets, written $\llbracket cs \rrbracket$ where cs is a set of characters. In applications this regular expression is often written as $[A{-}Z]$ to stand for example for the regular expression that can match any capital letter, or as $[\mathit{0}{-}\mathit{9}]$ to match any numeral. Such character sets can of course be represented by using alternatives (or $\mathbf{0}$ if the set is empty) and therefore do not add anything new in terms of recognised languages. We include them here

[2] An extended version of [4] is available at the website of its first author; this extended version already includes remarks in the appendix that their informal proof contains gaps, and possible fixes are not fully worked out.

because they will show later on that the generality of a definition is required once such simple regular expressions are added. They are also a 'low hanging fruit' in terms of improving the runtime of derivative matchers. The reason is that a simple membership test with the character sets can replace operations that otherwise need to traverse sizable regular expressions.

The language of a regular expression is defined as usual by the recursive function L with the seven clauses:

$$\begin{array}{lrcl}
\textit{(1)} & L(\mathbf{0}) & \overset{\text{def}}{=} & \varnothing \\
\textit{(2)} & L(\mathbf{1}) & \overset{\text{def}}{=} & \{[]\} \\
\textit{(3)} & L(c) & \overset{\text{def}}{=} & \{[c]\} \\
\textit{(4)} & L(r_1 \cdot r_2) & \overset{\text{def}}{=} & L(r_1) \mathbin{@} L(r_2) \\
\textit{(5)} & L(r_1 + r_2) & \overset{\text{def}}{=} & L(r_1) \cup L(r_2) \\
\textit{(6)} & L(r^*) & \overset{\text{def}}{=} & (L(r))* \\
\textit{(7)} & L([\![cs]\!]) & \overset{\text{def}}{=} & \{[c] \mid c \in cs\}
\end{array}$$

In clause *(4)* we use the operation _ @ _ for the concatenation of two languages (it is also list-append for strings). We use the star-notation for regular expressions and for languages (in the clause *(6)* above). The star for languages is defined inductively by two clauses: (i) the empty string being in the star of a language and (ii) if s_1 is in a language and s_2 in the star of this language, then also $s_1 \mathbin{@} s_2$ is in the star of this language. It will also be convenient to use the following notion of a *semantic derivative* (or *left quotient*) of a language defined as

$$\textit{Der}\ c\ A \overset{\textit{def}}{=} \{s \mid c :: s \in A\}\ .$$

For semantic derivatives we have the following equations (for example mechanically proved in [7]):

$$\begin{array}{ll}
\textit{Der}\ c\ \varnothing & \overset{\text{def}}{=} \varnothing \\
\textit{Der}\ c\ \{[]\} & \overset{\text{def}}{=} \varnothing \\
\textit{Der}\ c\ \{[d]\} & \overset{\text{def}}{=} \textit{if}\ c = d\ \textit{then}\ \{[]\}\ \textit{else}\ \varnothing \\
\textit{Der}\ c\ (A \cup B) & \overset{\text{def}}{=} \textit{Der}\ c\ A \cup \textit{Der}\ c\ B \\
\textit{Der}\ c\ (A \mathbin{@} B) & \overset{\text{def}}{=} (\textit{Der}\ c\ A \mathbin{@} B) \cup (\textit{if}\ [] \in A\ \textit{then}\ \textit{Der}\ c\ B\ \textit{else}\ \varnothing) \\
\textit{Der}\ c\ (A*) & \overset{\text{def}}{=} \textit{Der}\ c\ A \mathbin{@} A* \\
\textit{Der}\ c\ \{[c] \mid c \in cs\} & \overset{\text{def}}{=} \textit{if}\ c \in cs\ \textit{then}\ \{[]\}\ \textit{else}\ \varnothing
\end{array} \tag{1}$$

Brzozowski's derivatives of regular expressions [5] can be easily defined by two recursive functions: the first is from regular expressions to booleans (implementing a test when a regular expression can match the empty string), and the second takes a regular expression and a character to a (derivative) regular expression:

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

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

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

$$\begin{array}{lcl}
r\backslash[] & \stackrel{\text{def}}{=} & r \\
r\backslash(c :: s) & \stackrel{\text{def}}{=} & (r\backslash c)\backslash s
\end{array}$$

Given the equations in (1), it is a relatively easy exercise in mechanical reasoning to establish that

Proposition 1
(1) $\mathit{nullable}\ r$ *if and only if* $[] \in L(r)$, *and*
(2) $L(r\backslash c) = \mathit{Der}\ c\ (L(r))$.

With this in place it is also very routine to prove that the regular expression matcher defined as

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

gives a positive answer if and only if $s \in L(r)$. Consequently, this regular expression matching algorithm satisfies the usual specification for regular expression matching. While the matcher above calculates a provably correct YES/NO answer for whether a regular expression matches a string or not, the novel idea of Sulzmann and Lu [4] is to append another phase to this algorithm in order to calculate a lexical value. We will explain the details next.

3 POSIX Regular Expression Matching

There have been many previous works that use values for encoding *how* a regular expression matches a string. The clever idea by Sulzmann and Lu [4] is to define a function on values that mirrors (but inverts) the construction of the derivative on regular expressions. *Values* are defined as the inductive datatype

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

where we use vs to stand for a list of values. (This is similar to the approach taken by Frisch and Cardelli for GREEDY matching [10], and Sulzmann and Lu for POSIX matching [4]). The string underlying a value can be calculated by the *flat* function, written $|\_|$ and defined as:

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

We will sometimes refer to the underlying string of a value as *flattened value*. We will also overload our notation and use $|\mathit{vs}|$ for flattening a list of values and concatenating the resulting strings.

Sulzmann and Lu follow Nielsen and Henglein and define inductively an *inhabitation relation* that associates values to regular expressions (see [4, 15]). We define this relation as follows:[3]

$$\begin{array}{ccc}
\dfrac{}{\vdash \mathit{Empty} : \mathbf{1}} & \dfrac{}{\vdash \mathit{Char}\ c : c} & \dfrac{c \in cs}{\vdash \mathit{Char}\ c : [\![cs]\!]} \\[2ex]
\multicolumn{3}{c}{\dfrac{\vdash v_1 : r_1}{\vdash \mathit{Left}\ v_1 : r_1 + r_2} \qquad \dfrac{\vdash v_2 : r_1}{\vdash \mathit{Right}\ v_2 : r_2 + r_1}} \\[2ex]
\multicolumn{3}{c}{\dfrac{\vdash v_1 : r_1 \qquad \vdash v_2 : r_2}{\vdash \mathit{Seq}\ v_1\ v_2 : r_1 \cdot r_2} \qquad \dfrac{\forall\, v {\in} \mathit{vs}.\ \vdash v : r \wedge |v| \neq []}{\vdash \mathit{Stars}\ \mathit{vs} : r^*}}
\end{array} \tag{2}$$

where in the clause for *Stars* we use the notation $v \in \mathit{vs}$ for indicating that v is a member in the list *vs*. We require in this rule that every value in *vs* flattens to a non-empty string. The idea is that *Stars*-values satisfy the informal Star Rule (see Introduction) where the * does not match the empty string unless this is the only match for the repetition. That means for example the value

$$\mathit{Stars}\ [\mathit{Left}\ (\mathit{Char}\ a), \mathit{Empty}, \mathit{Right}\ (\mathit{Char}\ b)]$$

[3]Note that the rule for *Stars* differs from our earlier paper [1]. There we used the original definition by Sulzmann and Lu which does not require that the values $v \in \mathit{vs}$ flatten to a non-empty string (see also [15]). Our reason for introducing the more restricted version of lexical values is that the resulting set is always finite, given an r and s, which provides more convenience later on when reasoning about an ordering relation for values.

is *not* inhabited by the regular expression $(a + b)^*$, but the value

$$\mathit{Stars}\ [\mathit{Right}\ (\mathit{Char}\ b),\ \mathit{Right}\ (\mathit{Char}\ b)]$$

is. Note also that no values are associated with the regular expression $\mathbf{0}$ (since it does not match any string), and that the only value associated with the regular expression $\mathbf{1}$ is *Empty*. We use the *Char*-value for both, single character regular expressions and character sets. It is routine to establish how values "inhabiting" a regular expression correspond to the language of a regular expression, namely

Proposition 2 $L(r) = \{|v| \mid \vdash v : r\}$

Given a regular expression r and a string s, we define the set of all *Lexical Values* inhabited by r with the underlying string being s:[4]

$$\mathit{LV}\ r\ s \stackrel{\mathit{def}}{=} \{v \mid \vdash v : r \land |v| = s\}$$

The main property of $\mathit{LV}\ r\ s$ is that it is always finite.

Proposition 3 *finite* $(\mathit{LV}\ r\ s)$

This finiteness property does not hold in general if we remove the side-condition about $|v| \neq []$ in the *Stars*-rule above. For example using Sulzmann and Lu's less restrictive definition, $\mathit{LV}\ (\mathbf{1}^*)\ []$ would contain infinitely many values, but according to our more restricted definition only a single value, namely $\mathit{LV}\ (\mathbf{1}^*)\ [] = \{\mathit{Stars}\ []\}$.

If a regular expression r matches a string s, then generally the set $\mathit{LV}\ r\ s$ is not just a singleton set. In case of POSIX matching the problem is to calculate the unique lexical value that satisfies the (informal) POSIX rules from the Introduction. Graphically the POSIX value calculation algorithm by Sulzmann and Lu can be illustrated by the picture in Figure 1 where the path from the left to the right involving *derivatives*/*nullable* is the first phase of the algorithm (calculating successive Brzozowski's derivatives) and *mkeps*/*inj*, the path from right to left, the second phase. This picture shows the steps required when a regular expression, say r_1, matches the string $[a, b, c]$. We first build the three derivatives (according to a, b and c). We then use *nullable* to find out whether the resulting derivative regular expression r_4 can match the empty string. If yes, we call the function *mkeps* that produces a value v_4 for how r_4 can match the empty string (taking into account the POSIX constraints in case there are several ways). This function is defined by the clauses:

[4]Okui and Suzuki refer to our lexical values as *canonical values* in [2]. The notion of *non-problematic values* by Cardelli and Frisch [10] is related, but not identical to our lexical values.

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

Fig. 1 The two phases of the algorithm by Sulzmann & Lu [4], 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.

$$\begin{array}{lcl}
\mathit{mkeps}\ \mathbf{1} & \stackrel{\text{def}}{=} & \mathit{Empty} \\
\mathit{mkeps}\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{mkeps}\ r_2) \\
\mathit{mkeps}\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{\mathit{if}\ \mathit{nullable}\ r_1\ \mathit{then}\ \mathit{Left}\ (\mathit{mkeps}\ r_1)\ \mathit{else}\ \mathit{Right}\ (\mathit{mkeps}\ r_2)} \\
\mathit{mkeps}\ (r^\star) & \stackrel{\text{def}}{=} & \mathit{Stars}\ []
\end{array}$$

Note that this function needs only to be partially defined, namely only for regular expressions that are nullable. In case *nullable* fails, the string $[a,\ b,\ c]$ cannot be matched by r_1 and the null value *None* is returned by the algorithm. Note also how this function makes some subtle choices leading to a POSIX value: for example if an alternative regular expression, say $r_1 + r_2$, can match the empty string and furthermore r_1 can match the empty string, then we return a *Left*-value. The *Right*-value will only be returned if r_1 cannot match the empty string.

The most interesting idea from Sulzmann and Lu [4] is the construction of a value for how r_1 can match the string $[a,\ b,\ c]$ from the value how the last derivative, r_4 in Figure 1, can match the empty string. Sulzmann and Lu achieve this by stepwise "injecting back" the characters into the values thus inverting the operation of building derivatives, but on the level of values. The corresponding function, called *inj*, takes three arguments, a regular expression, a character and a value. For example in the first (or right-most) *inj*-step in Figure 1 the regular expression r_3, the character c from the last derivative step and v_4, which is the value corresponding to the derivative regular expression r_4. The result is the new value v_3. The final result of the algorithm is the value v_1. The *inj* function is defined by recursion on regular expressions and by analysing the shape of values (corresponding to the derivative regular expressions).

$$\begin{array}{llcl}
\textit{(1)} & \textit{inj}\ d\ c\ (\textit{Empty}) & \stackrel{\text{def}}{=} & \textit{Char}\ c \\
\textit{(2)} & \textit{inj}\ [\![cs]\!]\ c\ (\textit{Empty}) & \stackrel{\text{def}}{=} & \textit{Char}\ c \\
\textit{(3)} & \textit{inj}\ (r_1 + r_2)\ c\ (\textit{Left}\ v_1) & \stackrel{\text{def}}{=} & \textit{Left}\ (\textit{inj}\ r_1\ c\ v_1) \\
\textit{(4)} & \textit{inj}\ (r_1 + r_2)\ c\ (\textit{Right}\ v_2) & \stackrel{\text{def}}{=} & \textit{Right}\ (\textit{inj}\ r_2\ c\ v_2) \\
\textit{(5)} & \textit{inj}\ (r_1 \cdot r_2)\ c\ (\textit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} & \textit{Seq}\ (\textit{inj}\ r_1\ c\ v_1)\ v_2 \\
\textit{(6)} & \textit{inj}\ (r_1 \cdot r_2)\ c\ (\textit{Left}\ (\textit{Seq}\ v_1\ v_2)) & \stackrel{\text{def}}{=} & \textit{Seq}\ (\textit{inj}\ r_1\ c\ v_1)\ v_2 \\
\textit{(7)} & \textit{inj}\ (r_1 \cdot r_2)\ c\ (\textit{Right}\ v_2) & \stackrel{\text{def}}{=} & \textit{Seq}\ (\textit{mkeps}\ r_1)\ (\textit{inj}\ r_2\ c\ v_2) \\
\textit{(8)} & \textit{inj}\ (r^*)\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ \textit{vs})) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: \textit{vs})
\end{array}$$

To better understand what is going on in this definition it might be instructive to look first at the three sequence cases (clauses *(5)* – *(7)*). In each of these cases we need to construct an "injected value" for $r_1 \cdot r_2$. Because of the 'shape' of the regular expression, this must be a value of the form *Seq* _ _. Recall the clause of the *derivative*-function for sequence regular expressions:

$$(r_1 \cdot r_2)\backslash c \stackrel{\text{def}}{=} \textit{if nullable}\ r_1\ \textit{then}\ (r_1\backslash c) \cdot r_2 + (r_2\backslash c)\ \textit{else}\ (r_1\backslash c) \cdot r_2$$

Consider first the *else*-branch where the derivative is $(r_1\backslash c) \cdot r_2$. The corresponding value must therefore be of the form $\textit{Seq}\ v_1\ v_2$, which matches the left-hand side in clause *(5)* of *inj*. In the *if*-branch the derivative is an alternative, namely $(r_1\backslash c) \cdot r_2 + (r_2\backslash c)$. This means we either have to consider a *Left*- or *Right*-value. In case of the *Left*-value we know further it must be a value for a sequence regular expression. Therefore the pattern we match in the clause *(6)* is $\textit{Left}\ (\textit{Seq}\ v_1\ v_2)$, while in *(7)* it is just $\textit{Right}\ v_2$. One more interesting point is in the right-hand side of clause *(7)*: since in this case the regular expression r_1 does not "contribute" to matching the string, that means it only matches the empty string, we need to call *mkeps* in order to construct a value for how r_1 can match this empty string. A similar argument applies for why we can expect in the left-hand side of clause *(8)* that the value is of the form $\textit{Seq}\ v\ (\textit{Stars}\ \textit{vs})$—the derivative of a star is $(r\backslash c) \cdot r^*$. Finally, the reason for why we can ignore the first argument in clause *(1)* of *inj* is that it will only ever be called in cases where $c = d$, but the usual linearity restrictions in patterns do not allow us to build this constraint explicitly into our function definition.[5] Similarly the clause in *(2)* will only be called in cases where $c \in cs$ holds. Notable in this clause, however, is the fact that we *cannot* ignore the second argument of the injection function (the character that is injected into the value), because otherwise there is no way to determine which character from the character set should be injected into the value.

The idea of the *inj*-function to "inject" a character, say c, into a value can be made precise by the first part of the following lemma, which shows that the underlying string of an injected value has a prepended character c; the second

[5]Sulzmann and Lu state this clause as $\textit{inj}\ c\ c\ (\textit{Empty}) \stackrel{\text{def}}{=} \textit{Char}\ c$, but our deviation is harmless.

part shows that the underlying string of an *mkeps*-value is always the empty string (given the regular expression is nullable since otherwise *mkeps* might not be defined).

Lemma 4
(1) *If* $\vdash v : r\backslash c$ *then* $|\mathit{inj}\ r\ c\ v| = c :: |v|$.
(2) *If* $\mathit{nullable}\ r$ *then* $|\mathit{mkeps}\ r| = []$.

Proof Both properties are by routine inductions: the first one can, for example, be proved by induction over the definition of *derivatives*; the second by an induction on r. There are no interesting cases. □

Having defined the *mkeps* and *inj* function we can extend Brzozowski's matcher from the Introduction so that a value is constructed (assuming the regular expression matches the string). The clauses of the Sulzmann and Lu lexer are

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

If the regular expression does not match the string, *None* is returned. If the regular expression *does* match the string, then *Some* value is returned. One important virtue of this algorithm is that it can be implemented with ease in any functional programming language and also in Isabelle/HOL. In the remaining part of this section we prove that this algorithm is correct.

The well-known idea of POSIX matching is informally defined by some rules such as the Longest Match and Priority Rules (see Introduction); as correctly argued in [4], this needs formal specification. Sulzmann and Lu define an "ordering relation" between values and argue that there is a maximum value, as given by the derivative-based algorithm. In contrast, we shall introduce a simple inductive definition that specifies directly what a *POSIX value* is, incorporating the POSIX-specific choices into the side-conditions of our rules. Our definition is inspired by the matching relation given by Vansummeren [13]. The relation we define is ternary and written as $(s, r) \to v$, relating strings, regular expressions and values; the inductive rules are given in Figure 2. We can prove that given a string s and regular expression r, the POSIX value v is uniquely determined by $(s, r) \to v$.

Theorem 5
(1) *If* $(s, r) \to v$ *then* $s \in L(r)$ *and* $|v| = s$.
(2) *If* $(s, r) \to v$ *and* $(s, r) \to v'$ *then* $v = v'$.

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

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

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

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

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

Fig. 2 Our inductive definition of POSIX values.

Proof Both by induction on the definition of $(s,\ r) \to v$. The second part follows by a case analysis of $(s,\ r) \to v'$ and the first part. □

We claim that our $(s,\ r) \to v$ relation captures the idea behind the four informal POSIX rules shown in the Introduction: Consider for example the rules $P{+}L$ and $P{+}R$ where the POSIX value for a string and an alternative regular expression, that is $(s,\ r_1 + r_2)$, is specified—it is always a *Left*-value, *except* when the string to be matched is not in the language of r_1; only then it is a *Right*-value (see the side-condition in $P{+}R$). Interesting is also the rule for sequence regular expressions (PS). The first two premises state that v_1 and v_2 are the POSIX values for $(s_1,\ r_1)$ and $(s_2,\ r_2)$ respectively. Consider now the third premise and note that the POSIX value of this rule should match the string $s_1\ @\ s_2$. According to the Longest Match Rule, we want that the s_1 is the longest initial split of $s_1\ @\ s_2$ such that s_2 is still recognised by r_2. Let us assume, contrary to the third premise, that there *exist* an s_3 and s_4 such that s_2 can be split up into a non-empty string s_3 and a possibly empty string s_4. Moreover the longer string $s_1\ @\ s_3$ can be matched by r_1 and the shorter s_4 can still be matched by r_2. In this case s_1 would *not* be the longest initial split of $s_1\ @\ s_2$ and therefore $\mathit{Seq}\ v_1\ v_2$ cannot be a POSIX value for $(s_1\ @\ s_2,\ r_1 \cdot r_2)$. The main point is that our side-condition ensures the Longest Match Rule is satisfied.

A similar condition is imposed on the POSIX value in the $P{*}$-rule. Also there we want that s_1 is the longest initial split of $s_1\ @\ s_2$ and furthermore the corresponding value v cannot be flattened to the empty string. In effect, we require that in each "iteration" of the star, some non-empty substring needs to be "chipped" away; only in case of the empty string we accept $\mathit{Stars}\ []$ as the POSIX value. Indeed we can show that our POSIX values are lexical values which exclude those *Stars* that contain subvalues that flatten to the empty string.

Lemma 6 *If* $(s,\, r) \rightarrow v$ *then* $v \in LV\; r\; s$.

Proof By routine induction on $(s,\, r) \rightarrow v$. □

Next is the lemma that shows the function *mkeps* calculates the POSIX value for the empty string and a nullable regular expression.

Lemma 7 *If* $nullable\; r$ *then* $([],\, r) \rightarrow mkeps\; r$.

Proof By routine induction on r. □

The central lemma for our POSIX relation is that the *inj*-function preserves POSIX values.

Lemma 8 *If* $(s,\, r\backslash c) \rightarrow v$ *then* $(c :: s,\, r) \rightarrow inj\; r\; c\; v$.

Proof By induction on r. We explain two cases.

- Case $r = r_1 + r_2$. There are two subcases, namely (a) $v = Left\; v'$ and $(s,\, r_1\backslash c) \rightarrow v'$; and (b) $v = Right\; v'$, $s \notin L(r_1\backslash c)$ and $(s,\, r_2\backslash c) \rightarrow v'$. In (a) we know $(s,\, r_1\backslash c) \rightarrow v'$, from which we can infer $(c :: s,\, r_1) \rightarrow inj\; r_1\; c\; v'$ by induction hypothesis and hence $(c :: s,\, r_1 + r_2) \rightarrow inj\; (r_1 + r_2)\; c\; (Left\; v')$ as needed. Similarly in subcase (b) where, however, in addition we have to use Proposition 1(2) in order to infer $c :: s \notin L(r_1)$ from $s \notin L(r_1\backslash c)$.
- Case $r = r_1 \cdot r_2$. There are three subcases:

  (a) $v = Left\; (Seq\; v_1\; v_2)$ and $nullable\; r_1$
  (b) $v = Right\; v_1$ and $nullable\; r_1$
  (c) $v = Seq\; v_1\; v_2$ and $\neg\; nullable\; r_1$

  For (a) we know $(s_1,\, r_1\backslash c) \rightarrow v_1$ and $(s_2,\, r_2) \rightarrow v_2$ as well as

  $$\nexists\, s_3\; s_4.a.\; s_3 \neq [] \wedge s_3 \;@\; s_4 = s_2 \wedge s_1 \;@\; s_3 \in L(r_1\backslash c) \wedge s_4 \in L(r_2)$$

  From the latter we can infer by Proposition 1(2):

  $$\nexists\, s_3\; s_4.a.\; s_3 \neq [] \wedge s_3 \;@\; s_4 = s_2 \wedge c :: s_1 \;@\; s_3 \in L(r_1) \wedge s_4 \in L(r_2)$$

  We can use the induction hypothesis for r_1 to obtain $(c :: s_1,\, r_1) \rightarrow inj\; r_1\; c\; v_1$. Putting this all together allows us to infer $(c :: s_1 \;@\; s_2,\, r_1 \cdot r_2) \rightarrow Seq\; (inj\; r_1\; c\; v_1)\; v_2$. The case (c) is similar.

For (b) we know $(s,\ r_2\backslash c) \rightarrow v_1$ and $s_1\ @\ s_2 \notin L((r_1\backslash c)\ \cdot\ r_2)$. From the former we have $(c :: s,\ r_2) \rightarrow \mathit{inj}\ r_2\ c\ v_1$ by induction hypothesis for r_2. From the latter we can infer

$$\nexists s_3\ s_4.a.\ s_3 \neq []\ \wedge\ s_3\ @\ s_4 = c :: s\ \wedge\ s_3 \in L(r_1)\ \wedge\ s_4 \in L(r_2)$$

By Lemma 7 we know $([],\ r_1) \rightarrow \mathit{mkeps}\ r_1$ holds. Putting this all together, we can conclude with $(c :: s,\ r_1\ \cdot\ r_2) \rightarrow \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{inj}\ r_2\ c\ v_1)$, as required.

Finally suppose $r = r_1{}^*$. This case is very similar to the sequence case, except that we need to also ensure that $|\mathit{inj}\ r_1\ c\ v_1| \neq []$. This follows from $(c :: s_1,\ r_1) \rightarrow \mathit{inj}\ r_1\ c\ v_1$ (which in turn follows from $(s_1,\ r_1\backslash c) \rightarrow v_1$ and the induction hypothesis).

□

With Lemma 8 in place, it is completely routine to establish that the Sulzmann and Lu lexer satisfies our specification (returning the null value *None* iff the string is not in the language of the regular expression, and returning a unique POSIX value iff the string *is* in the language):

Theorem 9

(1) $s \notin L(r)$ *if and only if* $\mathit{lexer}\ r\ s = \mathit{None}$
(2) $s \in L(r)$ *if and only if* $\exists v.\ \mathit{lexer}\ r\ s = \mathit{Some}\ v\ \wedge\ (s,\ r) \rightarrow v$

Proof By induction on s using Lemma 7 and 8. □

In *(2)* we further know by Theorem 5 that the value returned by the lexer must be unique. A simple corollary of our two theorems therefore is:

Corollary 10

(1) $\mathit{lexer}\ r\ s = \mathit{None}$ *if and only if* $\nexists v.a.\ (s,\ r) \rightarrow v$
(2) $\mathit{lexer}\ r\ s = \mathit{Some}\ v$ *if and only if* $(s,\ r) \rightarrow v$

This concludes our correctness proof. Note that we have not changed the algorithm of Sulzmann and Lu,[6] but introduced our own specification for what a correct result—a POSIX value—should be. In the next section we show that our specification coincides with another one given by Okui and Suzuki using a different technique.

[6] All deviations we introduced are harmless.

4 Ordering of Values according to Okui and Suzuki

While in the previous section we have defined POSIX values directly in terms of a ternary relation (see inference rules in Figure 2), Sulzmann and Lu took a different approach in [4]: they introduced an ordering for values and identified POSIX values as the maximal elements. An extended version of [4] is available at the website of its first author; this includes more details of their proofs, but which are evidently not in final form yet. Unfortunately, we were not able to verify claims that their ordering has properties such as being transitive or having maximal elements.

Okui and Suzuki [2, 3] described another ordering of values, which they use to establish the correctness of their automata-based algorithm for POSIX matching. Their ordering resembles some aspects of the one given by Sulzmann and Lu, but overall is quite different. To begin with, Okui and Suzuki identify POSIX values as minimal, rather than maximal, elements in their ordering. A more substantial difference is that the ordering by Okui and Suzuki uses *positions* in order to identify and compare subvalues. Positions are lists of natural numbers. This allows them to quite naturally formalise the Longest Match and Priority rules of the informal POSIX standard. Consider for example the value v

$$v \stackrel{\text{def}}{=} \mathit{Stars}\ [\mathit{Seq}\ (\mathit{Char}\ x)\ (\mathit{Char}\ y),\ \mathit{Char}\ z]$$

At position $[0,1]$ of this value is the subvalue *Char y* and at position $[1]$ the subvalue *Char z*. At the 'root' position, or empty list $[]$, is the whole value v. Positions such as $[0,1,0]$ or $[2]$ are outside of v. If it exists, the subvalue of v at a position p, written $v\downharpoonleft_p$, can be recursively defined by

$$\begin{array}{rcl}
v\downharpoonleft_{[]} & \stackrel{\text{def}}{=} & v\\
\mathit{Left}\ v\downharpoonleft_{0::ps} & \stackrel{\text{def}}{=} & v\downharpoonleft_{ps}\\
\mathit{Right}\ v\downharpoonleft_{1::ps} & \stackrel{\text{def}}{=} & v\downharpoonleft_{ps}\\
\mathit{Seq}\ v_1\ v_2\downharpoonleft_{0::ps} & \stackrel{\text{def}}{=} & v_1\downharpoonleft_{ps}\\
\mathit{Seq}\ v_1\ v_2\downharpoonleft_{1::ps} & \stackrel{\text{def}}{=} & v_2\downharpoonleft_{ps}\\
\mathit{Stars}\ vs\downharpoonleft_{n::ps} & \stackrel{\text{def}}{=} & vs_{[n]}\downharpoonleft_{ps}
\end{array}$$

In the last clause we use Isabelle's notation $vs_{[n]}$ for the nth element in a list. The set of positions inside a value v, written *Pos v*, is given by

$$\begin{array}{ll}
\mathit{Pos}\ (\mathit{Empty}) & \stackrel{\text{def}}{=} \{[]\}\\
\mathit{Pos}\ (\mathit{Char}\ c) & \stackrel{\text{def}}{=} \{[]\}\\
\mathit{Pos}\ (\mathit{Left}\ v) & \stackrel{\text{def}}{=} \{[]\} \cup \{0::ps \mid ps \in \mathit{Pos}\ v\}\\
\mathit{Pos}\ (\mathit{Right}\ v) & \stackrel{\text{def}}{=} \{[]\} \cup \{1::ps \mid ps \in \mathit{Pos}\ v\}\\
\mathit{Pos}\ (\mathit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} \{[]\} \cup \{0::ps \mid ps \in \mathit{Pos}\ v_1\} \cup \{1::ps \mid ps \in \mathit{Pos}\ v_2\}\\
\mathit{Pos}\ (\mathit{Stars}\ vs) & \stackrel{\text{def}}{=} \{[]\} \cup (\bigcup n < \mathit{len}\ vs\ \{n::ps \mid ps \in \mathit{Pos}\ vs_{[n]}\})
\end{array}$$

whereby *len* in the last clause stands for the length of a list. Clearly for every position inside a value there exists a subvalue at that position.

To help understanding the ordering of Okui and Suzuki, consider again the earlier value v and compare it with the following w:

$$\begin{array}{l} v \stackrel{\text{def}}{=} \mathit{Stars}\ [\mathit{Seq}\ (\mathit{Char}\ x)\ (\mathit{Char}\ y),\ \mathit{Char}\ z] \\ w \stackrel{\text{def}}{=} \mathit{Stars}\ [\mathit{Char}\ x,\ \mathit{Char}\ y,\ \mathit{Char}\ z] \end{array}$$

Both values match the string xyz, that means if we flatten these values at their respective root position, we obtain xyz. However, at position $[0]$, v matches xy whereas w matches only the shorter x. So according to the Longest Match Rule, we should prefer v, rather than w as POSIX value for string xyz (and corresponding regular expression). In order to formalise this idea, Okui and Suzuki introduce a measure for subvalues at position p, called the *norm* of v at position p. We can define this measure in Isabelle as an integer as follows

$$\|v\|_p \stackrel{\text{def}}{=} \mathit{if}\ p \in \mathit{Pos}\ v\ \mathit{then}\ \mathit{len}\ |v\downharpoonleft_p|\ \mathit{else}\ -1$$

where we take the length of the flattened value at position p, provided the position is inside v; if not, then the norm is -1. The default for outside positions is crucial for the POSIX requirement of preferring a *Left*-value over a *Right*-value (if they can match the same string—see the Priority Rule from the Introduction). For this consider

$$v \stackrel{\text{def}}{=} \mathit{Left}\ (\mathit{Char}\ x) \qquad \text{and} \qquad w \stackrel{\text{def}}{=} \mathit{Right}\ (\mathit{Char}\ x)$$

Both values match x. At position $[0]$ the norm of v is 1 (the subvalue matches x), but the norm of w is -1 (the position is outside w according to how we defined the 'inside' positions of *Left*- and *Right*-values). Of course at position $[1]$, the norms $\|v\|_{[1]}$ and $\|w\|_{[1]}$ are reversed, but the point is that subvalues will be analysed according to lexicographically ordered positions. According to this ordering, the position $[0]$ takes precedence over $[1]$ and thus also v will be preferred over w. The lexicographic ordering of positions, written $\_ \prec_{lex} \_$, can be conveniently formalised by three inference rules

$$\frac{}{[] \prec_{lex} p :: ps} \qquad \frac{p_1 < p_2}{p_1 :: ps_1 \prec_{lex} p_2 :: ps_2} \qquad \frac{ps_1 \prec_{lex} ps_2}{p :: ps_1 \prec_{lex} p :: ps_2}$$

With the norm and lexicographic order in place, we can state the key definition of Okui and Suzuki [2]: a value v_1 is *smaller at position* p than v_2, written $v_1 \prec_p v_2$, if and only if (i) the norm at position p is greater in v_1 (that is the string $|v_1\downharpoonleft_p|$ is longer than $|v_2\downharpoonleft_p|$) and (ii) all subvalues at positions that are inside v_1 or v_2 and that are lexicographically smaller than p, we have the same norm, namely

$$v_1 \prec_p v_2 \overset{def}{=} \begin{cases} (i) & \|v_2\|_p < \|v_1\|_p \quad \text{and} \\ (ii) & \forall\, q \in Pos\ v_1 \cup Pos\ v_2.\ q \prec_{lex} p \longrightarrow \|v_1\|_q = \|v_2\|_q \end{cases}$$

The position p in this definition acts as the *first distinct position* of v_1 and v_2, where both values match strings of different length [2]. Since at p the values v_1 and v_2 match different strings, the ordering is irreflexive. Derived from the definition above are the following two orderings:

$$\begin{aligned} v_1 \prec v_2 &\overset{def}{=} \exists\, p.\ v_1 \prec_p v_2 \\ v_1 \preccurlyeq v_2 &\overset{def}{=} v_1 \prec v_2 \lor v_1 = v_2 \end{aligned}$$

While we encountered a number of obstacles for establishing properties like transitivity for the ordering of Sulzmann and Lu (and which we failed to overcome), it is relatively straightforward to establish this property for the orderings $\_ \prec \_$ and $\_ \preccurlyeq \_$ by Okui and Suzuki.

Lemma 11 (Transitivity) *If* $v_1 \prec v_2$ *and* $v_2 \prec v_3$ *then* $v_1 \prec v_3$.

Proof From the assumption we obtain two positions p and q, where the values v_1 and v_2 (respectively v_2 and v_3) are 'distinct'. Since $\prec_{lex}$ is trichotomous, we need to consider three cases, namely $p = q$, $p \prec_{lex} q$ and $q \prec_{lex} p$. Let us look at the first case. Clearly $\|v_2\|_p < \|v_1\|_p$ and $\|v_3\|_p < \|v_2\|_p$ imply $\|v_3\|_p < \|v_1\|_p$. It remains to show that for a $p' \in Pos\ v_1 \cup Pos\ v_3$ with $p' \prec_{lex} p$ that $\|v_1\|_{p'} = \|v_3\|_{p'}$ holds. Suppose $p' \in Pos\ v_1$, then we can infer from the first assumption that $\|v_1\|_{p'} = \|v_2\|_{p'}$. But this means that p' must be in $Pos\ v_2$ too (the norm cannot be -1 given $p' \in Pos\ v_1$). Hence we can use the second assumption and infer $\|v_2\|_{p'} = \|v_3\|_{p'}$, which concludes this case with $v_1 \prec v_3$. The reasoning in the other cases is similar. □

The proof for $\preccurlyeq$ is similar and omitted. It is also straightforward to show that $\prec$ and $\preccurlyeq$ are partial orders, and $\prec$ is well-founded over lexical values of a given regular expression and given string. Okui and Suzuki furthermore show that they are linear orderings for lexical values [2], but we have not formalised this in Isabelle. It is not essential for our results. What we are going to show below is that for a given r and s, the orderings have a unique minimal element on the set $LV\ r\ s$, which is the POSIX value we defined in the previous section. We start with two properties that show how the length of a flattened value relates to the $\prec$-ordering.

Proposition 12

(1) *If* $v_1 \prec v_2$ *then* $len\ |v_2| \leq len\ |v_1|$.
(2) *If* $len\ |v_2| < len\ |v_1|$ *then* $v_1 \prec v_2$.

Both properties follow from the definition of the ordering. Note that *(2)* entails that a value, say v_2, whose underlying string is a strict prefix of another flattened value, say v_1, then v_1 must be smaller than v_2. For our proofs it will be useful to have the following properties—in each case the underlying strings of the compared values are the same:

Proposition 13

(1) $\mathit{If}\ |v_1| = |v_2|\ \mathit{then}\ \mathit{Left}\ v_1 \prec \mathit{Right}\ v_2.$
(2) $\mathit{If}\ |v_1| = |v_2|\ \mathit{then}\ \mathit{Left}\ v_1 \prec \mathit{Left}\ v_2\ \mathit{iff}\ v_1 \prec v_2$
(3) $\mathit{If}\ |v_1| = |v_2|\ \mathit{then}\ \mathit{Right}\ v_1 \prec \mathit{Right}\ v_2\ \mathit{iff}\ v_1 \prec v_2$
(4) $\mathit{If}\ |v_2| = |w_2|\ \mathit{then}\ \mathit{Seq}\ v\ v_2 \prec \mathit{Seq}\ v\ w_2\ \mathit{iff}\ v_2 \prec w_2$
(5) $\mathit{If}\ |v_1|\ @\ |v_2| = |w_1|\ @\ |w_2|\ \mathit{and}\ v_1 \prec w_1\ \mathit{then}\ \mathit{Seq}\ v_1\ v_2 \prec \mathit{Seq}\ w_1\ w_2$
(6) $\mathit{If}\ |vs_1| = |vs_2|\ \mathit{then}$
$\mathit{Stars}\ (vs\ @\ vs_1) \prec \mathit{Stars}\ (vs\ @\ vs_2)\ \mathit{iff}\ \mathit{Stars}\ vs_1 \prec \mathit{Stars}\ vs_2$
(7) $\mathit{If}\ |v_1 :: vs_1| = |v_2 :: vs_2|\ \mathit{and}\ v_1 \prec v_2\ \mathit{then}$
$\mathit{Stars}\ (v_1 :: vs_1) \prec \mathit{Stars}\ (v_2 :: vs_2)$

One might prefer that statements *(4)* and *(5)* (respectively *(6)* and *(7)*) are combined into a single *iff*-statement (like the ones for *Left* and *Right*). Unfortunately this cannot be done easily: such a single statement would require an additional assumption about the two values $\mathit{Seq}\ v_1\ v_2$ and $\mathit{Seq}\ w_1\ w_2$ being inhabited by the same regular expression. The complexity of the proofs involved seems to not justify such a 'cleaner' single statement. The statements given are just the properties that allow us to establish our theorems without any difficulty. The proofs for Proposition 13 are routine.

Next we establish how Okui and Suzuki's orderings relate to our definition of *POSIX* values. Given a *POSIX* value v_1 for r and s, then any other lexical value v_2 in $\mathit{LV}\ r\ s$ is greater or equal than v_1, namely:

Theorem 14 *If* $(s,\, r) \rightarrow v_1$ *and* $v_2 \in \mathit{LV}\ r\ s$ *then* $v_1 \preccurlyeq v_2$.

Proof By induction on our *POSIX* rules. By Theorem 5 and the definition of *LV*, it is clear that v_1 and v_2 have the same underlying string s. The four base cases are straightforward: for example for $v_1 = \mathit{Empty}$, we have that $v_2 \in \mathit{LV}\ \mathbf{1}\ []$ must also be of the form $v_2 = \mathit{Empty}$. Therefore we have $v_1 \preccurlyeq v_2$. The inductive cases for r being of the form $r_1 + r_2$ and $r_1 \cdot r_2$ are as follows:

- Case $P{+}L$ with $(s,\, r_1 + r_2) \rightarrow \mathit{Left}\ w_1$: In this case the value v_2 is either of the form $\mathit{Left}\ w_2$ or $\mathit{Right}\ w_2$. In the latter case we can immediately conclude with $v_1 \preccurlyeq v_2$ since a *Left*-value with the same underlying string s is always smaller than a *Right*-value by Proposition 13*(1)*. In the former case we have $w_2 \in \mathit{LV}\ r_1\ s$ and can use the induction hypothesis to infer $w_1 \preccurlyeq w_2$. Because w_1 and w_2 have the same underlying string s, we can conclude with $\mathit{Left}\ w_1 \preccurlyeq \mathit{Left}\ w_2$ using Proposition 13*(2)*.

- Case $P{+}R$ with $(s,\ r_1 + r_2) \to \mathit{Right}\ w_1$: This case similar to the previous case, except that we additionally know $s \notin L(r_1)$. This is needed when v_2 is of the form $\mathit{Left}\ w_2$. Since $|v_2| = |w_2| = s$ and $\vdash w_2 : r_1$, we can derive a contradiction for $s \notin L(r_1)$ using Proposition 2. So also in this case $v_1 \preccurlyeq v_2$.
- Case PS with $(s_1 @ s_2,\ r_1 \cdot r_2) \to \mathit{Seq}\ w_1\ w_2$: We can assume $v_2 = \mathit{Seq}\ u_1\ u_2$ with $\vdash u_1 : r_1$ and $\vdash u_2 : r_2$. We have $s_1 @ s_2 = |u_1| @ |u_2|$. By the side-condition of the PS-rule we know that either $s_1 = |u_1|$ or that $|u_1|$ is a strict prefix of s_1. In the latter case we can infer $w_1 \prec u_1$ by Proposition 12*(2)* and from this $v_1 \preccurlyeq v_2$ by Proposition 13*(5)* (as noted above v_1 and v_2 must have the same underlying string). In the former case we know $u_1 \in \mathit{LV}\ r_1\ s_1$ and $u_2 \in \mathit{LV}\ r_2\ s_2$. With this we can use the induction hypotheses to infer $w_1 \preccurlyeq u_1$ and $w_2 \preccurlyeq u_2$. By Proposition 13*(4,5)* we can again infer $v_1 \preccurlyeq v_2$.

The case for $P{*}$ is similar to the PS-case and omitted. □

This theorem shows that our *POSIX* value for a regular expression r and string s is in fact a minimal element of the values in $\mathit{LV}\ r\ s$. By Proposition 12*(2)* we also know that any value in $\mathit{LV}\ r\ s'$, with s' being a strict prefix, cannot be smaller than v_1. The next theorem shows the opposite—namely any minimal element in $\mathit{LV}\ r\ s$ must be a *POSIX* value. This can be established by induction on r, but the proof can be drastically simplified by using the fact from the previous section about the existence of a *POSIX* value whenever a string $s \in L(r)$.

Theorem 15 *If* $v_1 \in \mathit{LV}\ r\ s$ *and* $\forall\, v_2 \in \mathit{LV}\ r\ s.\ v_2 \not\prec v_1$ *then* $(s,\ r) \to v_1$.

Proof If $v_1 \in \mathit{LV}\ r\ s$ then $s \in L(r)$ by Proposition 2. Hence by Theorem 9(2) there exists a *POSIX* value v_P with $(s,\ r) \to v_P$ and by Lemma 6 we also have $v_P \in \mathit{LV}\ r\ s$. By Theorem 14 we therefore have $v_P \preccurlyeq v_1$. If $v_P = v_1$ then we are done. Otherwise we have $v_P \prec v_1$, which however contradicts the second assumption about v_1 being the smallest element in $\mathit{LV}\ r\ s$. So we are done in this case too. □

From this we can also show that if $\mathit{LV}\ r\ s$ is non-empty (or equivalently $s \in L(r)$) then it has a unique minimal element:

Corollary 16
If $\mathit{LV}\ r\ s \neq \varnothing$ *then* $\exists!\, v_{min}.\ v_{min} \in \mathit{LV}\ r\ s \wedge (\forall\, v \in \mathit{LV}\ r\ s.\ v_{min} \preccurlyeq v)$.

To sum up, we have shown that the (unique) minimal elements of the ordering by Okui and Suzuki are exactly the *POSIX* values we defined inductively in Section 3. This provides an independent confirmation that our ternary relation formalises the informal POSIX rules.

5 Optimisations

Derivatives as calculated by Brzozowski's method are usually more complex regular expressions than the initial one; the result is that the derivative-based matching and lexing algorithms are often abysmally slow. However, various optimisations are possible, such as the simplifications of $\mathbf{0} + r$, $r + \mathbf{0}$, $\mathbf{1} \cdot r$ and $r \cdot \mathbf{1}$ to r. These simplifications can speed up the algorithms considerably, as noted in [4]. One of the advantages of having a simple specification and correctness proof is that the latter can be refined to prove the correctness of such simplification steps. While the simplification of regular expressions according to rules like

$$\begin{array}{lcl} \mathbf{0} + r \Rightarrow r & \qquad & \mathbf{0} \cdot r \Rightarrow \mathbf{0} \\ r + \mathbf{0} \Rightarrow r & & r \cdot \mathbf{0} \Rightarrow \mathbf{0} \\ r + r \Rightarrow r & & \mathbf{1} \cdot r \Rightarrow r \\ & & r \cdot \mathbf{1} \Rightarrow r \end{array} \tag{3}$$

is well understood, there is an obstacle with the *POSIX* value calculation algorithm by Sulzmann and Lu: if we build a derivative regular expression and then simplify it, we will calculate a *POSIX* value for this simplified derivative regular expression, *not* for the original (unsimplified) derivative regular expression. Sulzmann and Lu [4] overcome this obstacle by not just calculating a simplified regular expression, but also calculating a *rectification function* that "repairs" the incorrect value.

The rectification functions can be (slightly clumsily) implemented in Isabelle/HOL as follows using some auxiliary functions:

$$\begin{array}{lcl}
F_{\mathit{Right}}\; f\; v & \stackrel{\text{def}}{=} & \mathit{Right}\; (f\; v) \\
F_{\mathit{Left}}\; f\; v & \stackrel{\text{def}}{=} & \mathit{Left}\; (f\; v) \\
F_{\mathit{Alt}}\; f_1\; f_2\; (\mathit{Right}\; v) & \stackrel{\text{def}}{=} & \mathit{Right}\; (f_2\; v) \\
F_{\mathit{Alt}}\; f_1\; f_2\; (\mathit{Left}\; v) & \stackrel{\text{def}}{=} & \mathit{Left}\; (f_1\; v) \\
F_{\mathit{Seq1}}\; f_1\; f_2\; v & \stackrel{\text{def}}{=} & \mathit{Seq}\; (f_1\; \mathit{Empty})\; (f_2\; v) \\
F_{\mathit{Seq2}}\; f_1\; f_2\; v & \stackrel{\text{def}}{=} & \mathit{Seq}\; (f_1\; v)\; (f_2\; \mathit{Empty}) \\
F_{\mathit{Seq}}\; f_1\; f_2\; (\mathit{Seq}\; v_1\; v_2) & \stackrel{\text{def}}{=} & \mathit{Seq}\; (f_1\; v_1)\; (f_2\; v_2) \\[1ex]
\mathit{simp}_{\mathit{Alt}}\; (\mathbf{0},\, \_)\; (r_2,\, f_2) & \stackrel{\text{def}}{=} & (r_2,\, F_{\mathit{Right}}\; f_2) \\
\mathit{simp}_{\mathit{Alt}}\; (r_1,\, f_1)\; (\mathbf{0},\, \_) & \stackrel{\text{def}}{=} & (r_1,\, F_{\mathit{Left}}\; f_1) \\
\mathit{simp}_{\mathit{Alt}}\; (r_1,\, f_1)\; (r_2,\, f_2) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{l}{\qquad \mathit{if}\; r_1 = r_2\; \mathit{then}\; (r_1,\, F_{\mathit{Left}}\; f_1)\; \mathit{else}\; (r_1 + r_2,\, F_{\mathit{Alt}}\; f_1\; f_2)} \\[1ex]
\mathit{simp}_{\mathit{Seq}}\; (\mathbf{0},\, \_)\; (r_2,\, \_) & \stackrel{\text{def}}{=} & (\mathbf{0},\, \mathit{undefined}) \\
\mathit{simp}_{\mathit{Seq}}\; (r_1,\, \_)\; (\mathbf{0},\, \_) & \stackrel{\text{def}}{=} & (\mathbf{0},\, \mathit{undefined}) \\
\mathit{simp}_{\mathit{Seq}}\; (\mathbf{1},\, f_1)\; (r_2,\, f_2) & \stackrel{\text{def}}{=} & (r_2,\, F_{\mathit{Seq1}}\; f_1\; f_2) \\
\mathit{simp}_{\mathit{Seq}}\; (r_1,\, f_1)\; (\mathbf{1},\, f_2) & \stackrel{\text{def}}{=} & (r_1,\, F_{\mathit{Seq2}}\; f_1\; f_2) \\
\mathit{simp}_{\mathit{Seq}}\; (r_1,\, f_1)\; (r_2,\, f_2) & \stackrel{\text{def}}{=} & (r_1 \cdot r_2,\, F_{\mathit{Seq}}\; f_1\; f_2)
\end{array}$$

The functions $simp_{Alt}$ and $simp_{Seq}$ encode the simplification rules in (3) and compose the rectification functions (simplifications can occur deep inside the regular expression). In the cases where we simplify regular expressions of the form $\mathbf{0} \cdot r$ and $r \cdot \mathbf{0}$ to just $\mathbf{0}$, we can use any rectification function because regular expression $\mathbf{0}$ will never lead to a successful match and hence the function will never be called. In Isabelle/HOL it is convenient to use *undefined* in such situations. The main simplification function is then

$$\begin{array}{lcl}
simp\ (r_1 + r_2) & \stackrel{\text{def}}{=} & simp_{Alt}\ (simp\ r_1)\ (simp\ r_2)\\
simp\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & simp_{Seq}\ (simp\ r_1)\ (simp\ r_2)\\
simp\ r & \stackrel{\text{def}}{=} & (r,\ id)
\end{array}$$

where *id* stands for the identity function. The function *simp* returns a simplified regular expression and a corresponding rectification function. Note that we do not simplify under stars: doing so seems to slow down the algorithm, rather than speed it up. The optimised lexer is then given by the clauses:

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

In the second clause we first calculate the derivative $r\backslash c$ and then simplify the result. This gives us a simplified derivative r_s and a rectification function f_r. The lexer is then recursively called with the simplified derivative, but before we inject the character c into the value v, we need to rectify v (that is construct $f_r\ v$). Before we can establish the correctness of $lexer^+$, we need to show that simplification preserves the language and simplification preserves our *POSIX* relation once the value is rectified (recall *simp* generates a (regular expression, rectification function) pair):

Lemma 17
(1) $L(fst\ (simp\ r)) = L(r)$
(2) *If* $(s,\ fst\ (simp\ r)) \rightarrow v$ *then* $(s,\ r) \rightarrow snd\ (simp\ r)\ v$.

Proof Both are by induction on r. There is no interesting case for the first statement. For the second statement, of interest are the $r = r_1 + r_2$ and $r = r_1 \cdot r_2$ cases. In each case we have to analyse subcases whether $fst\ (simp\ r_1)$ and $fst\ (simp\ r_2)$ equals $\mathbf{0}$ (respectively $\mathbf{1}$). For example for $r = r_1 + r_2$, consider the subcase $fst\ (simp\ r_1) = \mathbf{0}$ and $fst\ (simp\ r_2) \neq \mathbf{0}$. By assumption we know $(s,\ fst\ (simp\ (r_1 + r_2))) \rightarrow v$. From this we can infer $(s,\ fst\ (simp\ r_2)) \rightarrow v$ and by IH also (*) $(s,\ r_2) \rightarrow snd\ (simp\ r_2)\ v$. Given $fst\ (simp\ r_1) = \mathbf{0}$ we know $L(fst\ (simp\ r_1)) = \varnothing$. By the first statement $L(r_1)$ is the empty set, meaning (**) $s \notin L(r_1)$. Taking (*) and (**) together gives by the $P{+}R$-rule $(s,\ r_1 + r_2) \rightarrow \mathit{Right}\ (\mathit{snd}\ (\mathit{simp}\ r_2)\ v)$. In turn this gives $(s,\ r_1 + r_2) \rightarrow \mathit{snd}\ (\mathit{simp}\ (r_1 + r_2))\ v$ as we need to show. The other cases are similar. □

We can now prove relatively straightforwardly that the optimised lexer produces the expected result:

Theorem 18 $\mathit{lexer}^+\ r\ s = \mathit{lexer}\ r\ s$

Proof By induction on s generalising over r. The case $[]$ is trivial. For the cons-case suppose the string is of the form $c :: s$. By induction hypothesis we know $\mathit{lexer}^+\ r\ s = \mathit{lexer}\ r\ s$ holds for all r (in particular for r being the derivative $r\backslash c$). Let r_s be the simplified derivative regular expression, that is $\mathit{fst}\ (\mathit{simp}\ (r\backslash c))$, and f_r be the rectification function, that is $\mathit{snd}\ (\mathit{simp}\ (r\backslash c))$. We distinguish the cases whether (*) $s \in L(r\backslash c)$ or not. In the first case we have by Theorem 9(2) a value v so that $\mathit{lexer}\ (r\backslash c)\ s = \mathit{Some}\ v$ and $(s,\ r\backslash c) \rightarrow v$ hold. By Lemma 17(1) we can also infer from (*) that $s \in L(r_s)$ holds. Hence we know by Theorem 9(2) that there exists a v' with $\mathit{lexer}\ r_s\ s = \mathit{Some}\ v'$ and $(s,\ r_s) \rightarrow v'$. From the latter we know by Lemma 17(2) that $(s,\ r\backslash c) \rightarrow f_r\ v'$ holds. By the uniqueness of the *POSIX* relation (Theorem 5) we can infer that v is equal to $f_r\ v'$—that is the rectification function applied to v' produces the original v. Now the case follows by the definitions of *lexer* and lexer^+.

In the second case where $s \notin L(r\backslash c)$ we have that $\mathit{lexer}\ (r\backslash c)\ s = \mathit{None}$ by Theorem 9(1). We also know by Lemma 17(1) that $s \notin L(r_s)$. Hence $\mathit{lexer}\ r_s\ s = \mathit{None}$ by Theorem 9(1) and by IH then also $\mathit{lexer}^+\ r_s\ s = \mathit{None}$. With this we can conclude in this case too. □

6 Extensions

A strong point in favour of Sulzmann and Lu's algorithm is that it can be extended in various ways. If we are interested in tokenising a string, then we need to not just split up the string into tokens, but also "classify" the tokens (for example whether they are keywords or identifiers and so on). This can be done with only minor modifications to the algorithm by introducing *record regular expressions* and *record values* (for example [17]):

$$r := \ldots \mid (l : r) \qquad\qquad v := \ldots \mid (l : v)$$

where l is a label, say a string, r a regular expression and v a value. All functions can be smoothly extended to these regular expressions and values. For example $(l : r)$ is nullable iff r is, and so on. The purpose of the record regular expression is to mark certain parts of a regular expression and then record in the calculated value which parts of the string were matched by this part. The label can then serve as classification for the tokens. For this recall the regular expression $(r_{key} + r_{id})^*$ for keywords and identifiers from the Introduction. With the record regular expression we can form $((\mathit{key} : r_{key}) + (\mathit{id} : r_{id}))^*$ and then traverse the calculated value and only collect the underlying strings in record values. With this we obtain finite sequences of pairs of labels and strings, for example

$$(l_1 : s_1), ..., (l_n : s_n)$$

from which tokens with classifications (keyword-token, identifier-token and so on) can be extracted.

In the context of POSIX matching, it is also interesting to study additional constructors about bounded-repetitions of regular expressions. For this let us extend the results from the previous sections to the following four additional regular expression constructors:

$$\begin{array}{rcll} r & := & \ldots \mid & r^{\{n\}} & \text{exactly-}n\text{-times} \\ & & \mid & r^{\{..n\}} & \text{upto-}n\text{-times} \\ & & \mid & r^{\{n..\}} & \text{from-}n\text{-times} \\ & & \mid & r^{\{n..m\}} & \text{between-}nm\text{-times} \end{array}$$

We will call them *bounded regular expressions.* They can be used to specify how many times a regular expression should match. With the help of the power operator (definition omitted) for sets of strings, the languages recognised by these regular expression can be defined in Isabelle as follows:

$$\begin{array}{lcl} L(r^{\{n\}}) & \stackrel{\text{def}}{=} & L(r)^n \\ L(r^{\{..n\}}) & \stackrel{\text{def}}{=} & \bigcup_{i\in\{..n\}} . \, L(r)^i \\ L(r^{\{n..\}}) & \stackrel{\text{def}}{=} & \bigcup_{i\in\{n..\}} . \, L(r)^i \\ L(r^{\{n..m\}}) & \stackrel{\text{def}}{=} & \bigcup_{i\in\{n..m\}} . \, L(r)^i \end{array}$$

This definition implies that in the last clause $r^{\{n..m\}}$ matches no string in case $m < n$, because then the interval $\{n..m\}$ is empty.

While the language recognised by these regular expressions is straightforward, some care is needed for how to define the corresponding lexical values. First, with a slight abuse of language, we will (re)use values of the form *Stars vs* for values inhabited in bounded regular expressions. Second, we need to introduce inductive rules for extending our inhabitation relation shown in (2), from which we then derived our notion of lexical values. Given the rule for r^*, the rule for $r^{\{..n\}}$ just requires additionally that the length of the list of values must be smaller or equal to n, that is:

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

Like in the r^*-rule, we require with the left-premise that some non-empty part of the string is 'chipped' away by *every* value in *vs*, that means the corresponding values do not flatten to the empty string.

In the rule for $r^{\{n\}}$ (that is exactly-n-times r) we will require that the length of the list of values equals to n. But enforcing in this case that every of these n values 'chips' away some part of a string would be too strong. Therefore matters are bit more complicated in the rule for $r^{\{n\}}$. According to the informal POSIX rules we have to allow that there is an "initial segment" that needs to chip away some parts of the string, but if this segment is too short for satisfying the exactly-n-times constraint, it can be followed by a segment where every value flattens to the empty string. One way for expressing this constraint in Isabelle is by the rule:

$$\frac{\forall v \in vs_1.\ \vdash v : r \wedge |v| \neq [] \quad \forall v \in vs_2.\ \vdash v : r \wedge |v| = [] \quad \mathit{len}\,(vs_1 \mathbin{@} vs_2) = n}{\vdash \mathit{Stars}\,(vs_1 \mathbin{@} vs_2) : r^{\{n\}}}$$

The vs_1 is the initial segment with non-empty flattened values, whereas vs_2 is the segment where all values flatten to the empty string. This idea gets even more complicated for the $r^{\{n..\}}$ regular expression. The reason is that we need to distinguish the case where we use fewer repetitions than n. In this case we need to "fill" the end with values that match the empty string to obtain at least n repetitions. But in case we need more than n repetitions, then *all* values should match a non-empty string. This leads to two inhabitation rules for $r^{\{n..\}}$:

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

Note that these two rules "collapse" in case $n = 0$ to just the single rule given for r^* in the definition shown in (2). We have similar rules for the between-nm-times operator (omitted). These rules ensure that our definition for sets of lexical values $LV\,r\,s$ are still finite and also fits with the ordering given by Okui and Suzuki (which require minimal values over the sets $LV\,r\,s$).

Fortunately, the other definitions extend "smoother" to bounded repetitions. For example the rules for derivatives are:

$$\begin{array}{lcl}
r^{\{n\}}\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ n = 0\ \mathit{then}\ \mathbf{0}\ \mathit{else}\ (r\backslash c) \cdot r^{\{n-1\}} \\
r^{\{..n\}}\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ n = 0\ \mathit{then}\ \mathbf{0}\ \mathit{else}\ (r\backslash c) \cdot r^{\{..n-1\}} \\
r^{\{n..\}}\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ n = 0\ \mathit{then}\ (r\backslash c) \cdot r^*\ \mathit{else}\ (r\backslash c) \cdot r^{\{n-1..\}} \\
r^{\{n..m\}}\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ m < n\ \mathit{then}\ \mathbf{0} \\
 & & \mathit{else\ if}\ n = 0\ \mathit{then} \\
 & & \qquad \mathit{if}\ m = 0\ \mathit{then}\ \mathbf{0}\ \mathit{else}\ (r\backslash c) \cdot r^{\{..m-1\}} \\
 & & \mathit{else}\ (r\backslash c) \cdot r^{\{n-1..m-1\}}
\end{array}$$

For *mkeps* we need to generate the shortest list of values we can get "away with" given the boundedness constraints. This means for example in the case $r^{\{..n\}}$ we can return the empty list, like for stars. In the other cases we have to generate a list of exactly n copies of the *mkeps*-value, because n is the smallest number of repetitions required.

$$\begin{array}{lcl}
\textit{mkeps}\ (r^{\{..n\}}) & \stackrel{\text{def}}{=} & \textit{Stars}\ []\\
\textit{mkeps}\ (r^{\{n\}}) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{replicate}\ n\,(\textit{mkeps}\ r))\\
\textit{mkeps}\ (r^{\{n..\}}) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{replicate}\ n\,(\textit{mkeps}\ r))\\
\textit{mkeps}\ (r^{\{n..m\}}) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{replicate}\ n\,(\textit{mkeps}\ r))
\end{array}$$

In this definition we use Isabelle's *replicate*-function in order to generate a list of n copies of a value. The injection function also extends straightforwardly to the bounded regular expressions as follows:

$$\begin{array}{lcl}
\textit{inj}\ (r^{\{n\}})\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ \textit{vs})) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: \textit{vs})\\
\textit{inj}\ (r^{\{n..\}})\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ \textit{vs})) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: \textit{vs})\\
\textit{inj}\ (r^{\{..n\}})\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ \textit{vs})) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: \textit{vs})\\
\textit{inj}\ (r^{\{n..m\}})\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ \textit{vs})) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: \textit{vs})
\end{array}$$

Similarly our POSIX definition can be easily extended to the additional constructors. For example for $r^{\{n\}}$ we have two rules:

$$\frac{\forall v \in vs.\ ([], r) \to v \qquad \textit{len}\ \textit{vs} = n}{([], r^{\{n\}}) \to \textit{Stars}\ \textit{vs}}$$

$$\frac{\begin{array}{c}(s_1,\ r) \to v \qquad (s_2,\ r^{\{n-1\}}) \to\ \textit{Stars}\ \textit{vs} \qquad |v| \neq [] \qquad 0 < n\\ \nexists s_3\ s_4.\ \ s_3 \neq []\ \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L(r) \wedge s_4 \in L(r^{\{n-1\}})\end{array}}{(s_1 @ s_2,\ r^{\{n\}}) \to \textit{Stars}\ (v :: \textit{vs})}$$

The first rule deals with the case when an empty string needs to be recognised. The second when the string is non-empty. In this case the "initial segment" must match non-empty strings only. The idea behind this formulation is to avoid situations where an earlier value matches the empty string, while it is actually possible to "nibble away" some parts of the string. The rules for the other bounded regular expressions are similar. We shall omit them here. With these definitions in place, our proofs given in the previous sections extend to the bounded repetitions. The main point is that there are no surprises.

What is good about our re-use of the *Stars*-constructor for the values of bounded regular expressions is that we did not need to make any changes to the ordering definitions by Okui and Suzuki. It still holds that our POSIX values are the minimal elements for the lexical value sets, and vice versa. In this way we again obtain independent assurance that our definitions capture correctly the idea behind POSIX matching.

Unfortunately, in our formal proofs in Isabelle/HOL we need to give the definitions and proofs all over again in a separate theory, since there is no way of making Isabelle to accept proofs for the basic regular expressions (defined as inductive datatype) and then augmenting the datatype with new constructors. This would be a really “cool” feature for Isabelle, but we have no idea how this could be achieved elegantly.

7 Conclusion

We have implemented in Isabelle/HOL the POSIX value calculation algorithm introduced by Sulzmann and Lu [4]. Our implementation is nearly identical to the original and all modifications we introduced are harmless (like our char-clause for *inj*). We have proved this algorithm to be correct, but correct according to our own specification of what POSIX values are. Our specification (inspired from work by Vansummeren [13]) appears to be much simpler than in [4] and our proofs are nearly always straightforward. As we were able to show, the work extends also to bounded regular expressions, character classes and records. We have attempted to formalise the original proof by Sulzmann and Lu [4], but we believe it contains unfillable gaps. In the online version of [4], the authors already acknowledge some small problems, but our experience suggests that there are more serious problems. We also showed that our definition for POSIX values is equivalent to a definition of POSIX values given by Okui and Suzuki [2]. They use a different technique for identifying POSIX values. This equivalence gives additional weight to our claim that our rules capture the informal ideas for POSIX lexing given in [11].

Having proved the correctness of the POSIX lexing algorithm in [4], which lessons have we learned? Well, this is a perfect example for the importance of the *right* definitions. We have (on and off) explored mechanisations as soon as first versions of [4] appeared, but have made little progress with turning the relatively detailed proof sketch in [4] into a formaliseable proof. Having seen [13] and adapted the POSIX definition given there for the algorithm by Sulzmann and Lu made all the difference: the proofs, as said, are nearly straightforward. The question remains whether the original proof idea of [4], potentially using our result as a stepping stone, can be made to work? Alas, we really do not know despite considerable effort.

In the context of formalising lexers in theorem provers, closely related to our work is an automata-based lexer formalised in Isabelle/HOL by Nipkow [18]. This lexer also splits up strings into longest initial substrings, but Nipkow's algorithm is not completely computational. The algorithm by Sulzmann and Lu, in contrast, can be implemented with ease in any functional language. A bespoke and executable lexer for the Imp-language is formalised in Coq as part of the Software Foundations book by Pierce et al [19]. The disadvantage of such bespoke lexers is that they do not generalise easily to more advanced features. Asperti et al [20, 21] formalise in the Matita theorem prover the notion of pointed regular expressions in order to elegantly generate DFAs for regular expression matching. While this work focuses on lexing using automata, we find most interesting the connection between the pointed regular expressions and Brzozowski derivatives. This might open up further work on calculating derivatives efficiently. We leave this to future work.

Most closely related to our work is the work by Egolf et al [22] on the Verbatim lexer, which is formalised in Coq. They have a similar inductive relation for specifying what POSIX matching means. What is good about their approach is that they calculate tokens directly; we in contrast have to use the record regular expression for this. Our approach is slightly more general, but this generality might not be wanted in typical applications. The authors of Verbatim report a good running time with one set of lexing rules, but it is unlikely that this good running time applies universally to all regular expressions and all strings. The reason is that they do not simplify derivatives, which means their sizes can explode. This is the main difference between their work and our work where we have shown that simplification rules do not affect the correctness of the algorithm by Sulzmann and Lu. A simple example where simplification makes a difference is the regular expression $(a^*)^* \cdot b$ whose derivatives can grow beyond any finite bound given long enough strings composed of just a's. In contrast, all derivatives of this regular expression stay below the size of 8 if they are simplified after each step. This is important because functions like nullable and derivative need to traverse regular expressions—if the size of derivatives is too large, then these functions will be slow, abysmally slow that is. There is also work by the same authors on Verbatim++, which is an improvement of the Verbatim lexer (using for example memoization) [23]. However, this work has a different focus than ours: their work uses derivatives in order to generate DFAs which are then used for lexing. While this might make the process of lexing faster for the "basic" regular expressions, classic DFAs have problems with bounded regular expressions. For them one has to connect many copies of DFAs, which increases their size and thus slows down the lexing process. As has been shown in this paper, derivatives can easily accomodate the bounded regular expressions without having to resort to constructing large DFAs.

Most recently the work by Moseley et al [24] has been included in the .NET7 regular expression library. They impressively extend Brzozowski derivatives to various *anchors* (like start-of-line or end-of-string) and *lookarounds* (like what is coming before or after a matched string). The latter has also been studied by Miyazaki and Minamide [25]. Moseley et al already mention a difference between their work and the work described here, namely that properties like $L(r_1 \cdot r_2) = L(r_1) \;@\; L(r_2)$ do not hold anymore when anchors are added. Another difference between their work and ours is that POSIX lexing is an inherently *asymmetric* problem, in the sense that it generates longest submatches (recall the Longest Match Rule from the Introduction). This is important for their matching algorithm where they define a reverse operator for regular expressions, written r^r, such that the following property holds:

$$L(r) = \{rev(s) \mid s \in L(r^r)\}$$

This means the language of the *reverse* regular expression is the set of reversed strings of $L(r)$. This property is useful for finding substring matches as it allows Moseley et al to first find the end-location where a substring matches a regular expression and then use $\_^r$ in order to find the beginning of the matched substring. The problem with POSIX lexing is that one cannot use the POSIX value for r^r and a string $rev(s)$ in order to generate the POSIX value for r and s. We leave a full investigation of what we can adopt from their work to future work.

Our formalisation is available from the Archive of Formal Proofs [26] under http://www.isa-afp.org/entries/Posix-Lexing.shtml.

Acknowledgements: I am very grateful to Martin Sulzmann for his comments on this work and moreover for patiently explaining the details in [4]. I am also very grateful to Fahad Ausaf who helped with this work and to Flavio Melinte Citea who pointed out omissions in the simplification rules.

I am deeply saddened that Roy Dyckhoff, co-author of the original conference paper [1] and the supervisor of my master thesis in St Andrews, died in August 2018. This was the last scientific paper he worked on. Roy was a witty, extremely intelligent and a very pleasant researcher and friend. He is much missed by me and many colleagues.

References

[1] Ausaf, F., Dyckhoff, R., Urban, C.: POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In: Proc. of the 7th International Conference on Interactive Theorem Proving (ITP). LNCS, vol. 9807, pp. 69–86 (2016)

[2] Okui, S., Suzuki, T.: 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). LNCS, vol. 6482, pp. 231–240 (2010)

[3] Okui, S., Suzuki, T.: Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. Technical report, University of Aizu (2013)

[4] Sulzmann, M., Lu, K.: POSIX Regular Expression Parsing with Derivatives. In: Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS). LNCS, vol. 8475, pp. 203–220 (2014)

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

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

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

[8] Coquand, T., Siles, V.: A Decision Procedure for Regular Expression Equivalence in Type Theory. In: Proc. of the 1st International Conference on Certified Programs and Proofs (CPP). LNCS, vol. 7086, pp. 119–134 (2011)

[9] Ribeiro, R., Bois, A.D.: Certified Bit-Coded Regular Expression Parsing. In: Proc. of the 21st Brazilian Symposium on Programming Languages. Association for Computing Machinery, New York, NY, USA (2017)

[10] Frisch, A., Cardelli, L.: Greedy Regular Expression Matching. In: Proc. of the 31st International Conference on Automata, Languages and Programming (ICALP). LNCS, vol. 3142, pp. 618–629 (2004)

[11] The Open Group Base Specification Issue 6 IEEE Std 1003.1 2004 Edition. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html (2004)

[12] Kuklewicz, C.: Regex Posix. https://wiki.haskell.org/Regex_Posix

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

[14] Hosoya, H., Vouillon, J., Pierce, B.C.: Regular Expression Types for XML. ACM Transactions on Programming Languages and Systems (TOPLAS) **27**(1), 46–90 (2005)

[15] Nielsen, L., Henglein, F.: Bit-Coded Regular Expression Parsing. In: Proc. of the 5th International Conference on Language and Automata Theory and Applications (LATA). LNCS, vol. 6638, pp. 402–413 (2011)

[16] Grathwohl, N.B.B., Henglein, F., Rasmussen, U.T.: A Crash-Course in Regular Expression Parsing and Regular Expressions as Types. Technical report, University of Copenhagen (2014)

[17] Sulzmann, M., van Steenhoven, P.: A Flexible and Efficient ML Lexer Tool Based on Extended Regular Expression Submatching. In: Proc. of the 23rd International Conference on Compiler Construction (CC). LNCS, vol. 8409, pp. 174–191 (2014)

[18] Nipkow, T.: Verified Lexical Analysis. In: Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 1479, pp. 1–15 (1998)

[19] Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software Foundations. Electronic textbook, ??? (2015). http://www.cis.upenn.edu/~bcpierce/sf

[20] Asperti, A., Coen, C.S., Tassi, E.: Regular expressions, au point (2010) https://arxiv.org/abs/1010.2604

[21] Asperti, A.: A Compact Proof of Decidability for Regular Expression Equivalence. In: Proc. of the 3rd International Conference on Interactive Theorem Proving (ITP). LNCS, vol. 7406, pp. 283–298 (2012)

[22] Egolf, D., Lasser, S., Fisher, K.: Verbatim: A Verified Lexer Generator. In: 2021 IEEE Security and Privacy Workshops (SPW), pp. 92–100 (2021)

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

[24] Moseley, D., Nishio, M., Perez Rodriguez, J., Saarikivi, O., Toub, S., Veanes, M., Wan, T., Xu, E.: Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. In: Proc. 44th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2023). (To appear.)

[25] Miyazaki, T., Minamide, Y.: Derivatives of Regular Expressions with Lookahead. Journal of Information Processing **27**, 422–430 (2019)

[26] Ausaf, F., Dyckhoff, R., Urban, C.: POSIX Lexing with Derivatives of Regular Expressions. Archive of Formal Proofs (2016). http://www.isa-afp.org/entries/Posix-Lexing.shtml, Formal proof development