POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl)

Fahad Ausaf[1], Roy Dyckhoff[2], and Christian Urban[3]

[1] King's College London
fahad.ausaf@icloud.com
[2] University of St Andrews
roy.dyckhoff@st-andrews.ac.uk
[3] King's College London
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. Sulzmann and Lu have made available on-line what they call a "rigorous proof" of the correctness of their algorithm w.r.t. their specification; regrettably, it appears to us to have unfillable gaps. In the first part of this paper we give our inductive definition of what a POSIX value is and show (i) that such a value is unique (for given regular expression and string being matched) and (ii) that Sulzmann and Lu's algorithm always generates such a value (provided that the regular expression matches the string). We also prove the correctness of an optimised version of the POSIX matching algorithm. Our definitions and proof are much simpler than those by Sulzmann and Lu and can be easily formalised in Isabelle/HOL. In the second part we analyse the correctness argument by Sulzmann and Lu and explain why the gaps in this argument cannot be filled easily.

Keywords: POSIX matching, Derivatives of Regular Expressions, Isabelle/HOL

1 Introduction

Brzozowski [1] 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 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 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 [8]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [5]. And another one in Coq is given by Coquand and Siles [2].

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 [3] and the other is POSIX matching [6,10,12]. For example consider the string xy and the regular expression $(x + y + xy)^\star$. Either the string can be matched in two 'iterations' by the single letter-regular expressions x and y, or directly in one iteration by xy. The first case corresponds to GREEDY matching, which first matches with the left-most symbol and only matches the next symbol in case of a mismatch (this is greedy in the sense of preferring instant gratification to delayed repletion). The second case is POSIX matching, which prefers the longest match.

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 two underlying (informal) rules behind tokenising a string in a POSIX fashion according to a collection of regular expressions:

- *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 regular expression that can match determines the token.

Consider for example a regular expression r_{key} for recognising keywords such as *if*, *then* and so on; and r_{id} recognising identifiers (say, a single character followed by characters or numbers). Then we can form the regular expression $(r_{key} + r_{id})^\star$ 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 a keyword token, not an identifier token—even if r_{id} matches also.

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 [10] extended this matcher to allow generation not just of a YES/NO answer but of an actual matching, called a [lexical] *value*. They 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 [10] 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 [3] 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. In Section 5 we identify some inherent problems with their approach (of which some of the proofs

are not published in [10]); perhaps more importantly, we give 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 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 [6] who found that nearly all POSIX matching implementations are "buggy" [10, Page 203] and by Grathwohl et al [4, 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 [10]. We have proved the correctness of this algorithm according to our specification of what a POSIX value is (inspired by work of Vansummeren [12]). Sulzmann and Lu sketch in [10] an informal correctness proof: but to us it contains unfillable gaps.[4] Our specification of a POSIX value consists of a simple inductive definition that given a string and a regular expression uniquely determines this value. Derivatives as calculated by Brzozowski's method are usually more complex regular expressions than the initial one; various optimisations are possible. We prove the correctness when simplifications of $\mathbf{0} + r$, $r + \mathbf{0}$, $\mathbf{1} \cdot r$ and $r \cdot \mathbf{1}$ to r are applied.

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 $[c]$. 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:

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

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

$$\begin{array}{lll}
(1) & L(\mathbf{0}) \stackrel{\text{def}}{=} \varnothing & (4) \quad L(r_1 \cdot r_2) \stackrel{\text{def}}{=} L(r_1) \mathbin{@} L(r_2) \\
(2) & L(\mathbf{1}) \stackrel{\text{def}}{=} \{[]\} & (5) \quad L(r_1 + r_2) \stackrel{\text{def}}{=} L(r_1) \cup L(r_2) \\
(3) & L(c) \stackrel{\text{def}}{=} \{[c]\} & (6) \quad L(r^\star) \stackrel{\text{def}}{=} (L(r))\star
\end{array}$$

[4] An extended version of [10] 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.

In clause (4) we use the operation $\_ \mathbin{@} \_$ 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 last clause 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

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

For semantic derivatives we have the following equations (for example mechanically proved in [5]):

$$\begin{array}{lcl} Der\ c\ \varnothing & \stackrel{\text{def}}{=} & \varnothing \\ Der\ c\ \{[]\} & \stackrel{\text{def}}{=} & \varnothing \\ Der\ c\ \{[d]\} & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \{[]\}\ \mathit{else}\ \varnothing \\ Der\ c\ (A \cup B) & \stackrel{\text{def}}{=} & Der\ c\ A \cup Der\ c\ B \\ Der\ c\ (A \mathbin{@} B) & \stackrel{\text{def}}{=} & (Der\ c\ A \mathbin{@} B) \cup (\mathit{if}\ [] \in A\ \mathit{then}\ Der\ c\ B\ \mathit{else}\ \varnothing) \\ Der\ c\ (A\star) & \stackrel{\text{def}}{=} & Der\ c\ A \mathbin{@} A\star \end{array} \tag{1}$$

Brzozowski's derivatives of regular expressions [1] 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:

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

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

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

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

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

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 [10] 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

The clever idea by Sulzmann and Lu [10] is to define values for encoding *how* a regular expression matches a string and then define a function on values that mirrors (but inverts) the construction of the derivative on regular expressions. *Values* are defined as the inductive datatype

$$v := () \mid Char\ c \mid Left\ v \mid Right\ v \mid Seq\ v_1\ v_2 \mid Stars\ vs$$

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 [3], and Sulzmann and Lu for POSIX matching [10]). The string underlying a value can be calculated by the *flat* function, written $|\_|$ and defined as:

$$\begin{array}{lcllcl}
|()| & \stackrel{\text{def}}{=} & [] & |Seq\ v_1\ v_2| & \stackrel{\text{def}}{=} & |v_1| \mathbin{@} |v_2| \\
|Char\ c| & \stackrel{\text{def}}{=} & [c] & |Stars\ []| & \stackrel{\text{def}}{=} & [] \\
|Left\ v| & \stackrel{\text{def}}{=} & |v| & |Stars\ (v :: vs)| & \stackrel{\text{def}}{=} & |v| \mathbin{@} |Stars\ vs| \\
|Right\ v| & \stackrel{\text{def}}{=} & |v| & & &
\end{array}$$

Sulzmann and Lu also define inductively an inhabitation relation that associates values to regular expressions:

$$\frac{}{() : \mathbf{1}} \qquad \frac{}{Char\ c : c}$$

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

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

$$\frac{}{Stars\ [] : r^\star} \qquad \frac{v : r \qquad Stars\ vs : r^\star}{Stars\ (v :: vs) : r^\star}$$

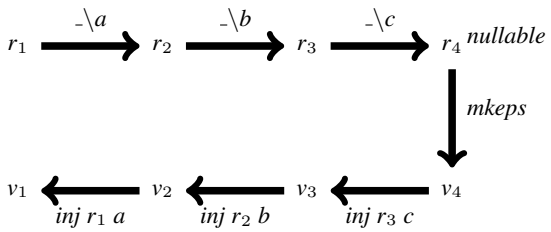

Fig. 1. The two phases of the algorithm by Sulzmann & Lu [10], matching the string $[a, b, c]$. The first phase (the arrows from left to right) is Brzozowski's matcher building successive derivatives. If the last regular expression is *nullable*, then the functions of the second phase are called (the top-down and right-to-left arrows): first *mkeps* calculates a value v_4 witnessing how the empty string has been recognised by r_4. After that the function *inj* "injects back" the characters of the string into the values.

Note that no values are associated with the regular expression $\mathbf{0}$, and that the only value associated with the regular expression $\mathbf{1}$ is $()$, pronounced (if one must) as *Void*. It is routine to establish how values "inhabiting" a regular expression correspond to the language of a regular expression, namely

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

In general there is more than one value associated with a regular expression. In case of POSIX matching the problem is to calculate the unique 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:

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

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. 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 [10] 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 Fig. 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 Fig. 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).

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

To better understand what is going on in this definition it might be instructive to look first at the three sequence cases (clauses (4)–(6)). In each case we need to construct an "injected value" for $r_1 \cdot r_2$. This must be a value of the form $\mathit{Seq}\ \_\ \_$. Recall the clause of the *derivative*-function for sequence regular expressions:

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

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 $\mathit{Seq}\ v_1\ v_2$, which matches the left-hand side in clause (4) 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 (5) is $\mathit{Left}\ (\mathit{Seq}\ v_1\ v_2)$, while in (6) it is just $\mathit{Right}\ v_2$. One more interesting point is in the right-hand side of clause (6): 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 (7) that the value is of the form $\mathit{Seq}\ v\ (\mathit{Stars}\ vs)$—the derivative of a star is $(r\backslash c) \cdot r^\star$. Finally, the reason for why we can ignore the second 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]

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 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 1.

(1) If $v : r\backslash c$ *then* $|inj\ r\ c\ v| = c :: |v|$.
(2) If $nullable\ r$ *then* $|mkeps\ r| = []$.

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 so that a [lexical] value is constructed (assuming the regular expression matches the string). The clauses of the Sulzmann and Lu lexer are

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

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 the longest match and priority rule (see Introduction); as correctly argued in [10], 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 [12]. The relation we define is ternary and written as $(s, r) \to v$, relating strings, regular expressions and values.

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

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

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

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

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

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

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 1.
(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* $v = v'$.

Proof. Both by induction on the definition of $(s, r) \to v$. The second parts 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 two 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 $s_1 \mathbin{@} s_2$. According to the longest match rule, we want that the s_1 is the longest initial split of $s_1 \mathbin{@} s_2$ 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 $s_1 \mathbin{@} s_3$ 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 $s_1 \mathbin{@} s_2$ and therefore $\mathit{Seq}\ v_1\ v_2$ cannot be a POSIX value for $(s_1 \mathbin{@} s_2, r_1 \cdot r_2)$. 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 $P\star$-rule. Also there we want that s_1 is the longest initial split of $s_1 \mathbin{@} s_2$ 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.

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

Lemma 2. *If nullable r then* $([], r) \to mkeps\ r$.

Proof. By routine induction on r. □

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

Lemma 3. *If* $(s, r\backslash c) \to v$ *then* $(c :: s, r) \to inj\ r\ c\ v$.

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 $(s, r_1\backslash c) \to v'$; and (b) $v = Right\ v'$, $s \notin L(r_1\backslash c)$ and $(s, r_2\backslash c) \to v'$. In (a) we know $(s, r_1\backslash c) \to v'$, from which we can infer $(c :: s, r_1) \to inj\ r_1\ c\ v'$ by induction hypothesis and hence $(c :: s, r_1 + r_2) \to inj\ (r_1 + r_2)\ c\ (Left\ v')$ as needed. Similarly in subcase (b) where, however, in addition we have to use Prop. 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 $(s_1, r_1\backslash c) \to v_1$ and $(s_2, r_2) \to v_2$ as well as

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

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

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

  We can use the induction hypothesis for r_1 to obtain $(c :: s_1, r_1) \to inj\ r_1\ c\ v_1$. Putting this all together allows us to infer $(c :: s_1 \mathbin{@} s_2, r_1 \cdot r_2) \to Seq\ (inj\ r_1\ c\ v_1)\ v_2$. The case (c) is similar.
  For (b) we know $(s, r_2\backslash c) \to v_1$ and $s_1 \mathbin{@} s_2 \notin L((r_1\backslash c) \cdot r_2)$. From the former we have $(c :: s, r_2) \to inj\ r_2\ c\ v_1$ by induction hypothesis for r_2. From the latter we can infer

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

  By Lem. 2 we know $([], r_1) \to mkeps\ r_1$ holds. Putting this all together, we can conclude with $(c :: s, r_1 \cdot r_2) \to Seq\ (mkeps\ r_1)\ (inj\ r_2\ c\ v_1)$, as required.
  Finally suppose $r = r_1^\star$. This case is very similar to the sequence case, except that we need to also ensure that $|inj\ r_1\ c\ v_1| \neq []$. This follows from $(c :: s_1, r_1) \to inj\ r_1\ c\ v_1$ (which in turn follows from $(s_1, r_1\backslash c) \to v_1$ and the induction hypothesis). □

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

Proof. By induction on s using Lem. 2 and 3. □

In (2) we further know by Thm. 1 that the value returned by the lexer must be unique. A simple corollary of our two theorems yis:

Corollary 1.
(1) $lexer\ r\ s = None$ *if and only if* $\nexists v.\ (s, r) \rightarrow v$
(2) $lexer\ r\ s = Some\ v$ *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. A strong point in favour of Sulzmann and Lu's algorithm is that it can be extended in various ways.

4 Extensions and Optimisations

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 it is a keyword or an identifier). This can be done with only minor modifications to the algorithm by introducing *record regular expressions* and *record values* (for example [11]):

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

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 $(r_{key} + r_{id})^\star$ for keywords and identifiers from the Introduction. With the record regular expression we can form $((key : r_{key}) + (id : r_{id}))^\star$ 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

$$(l_1 : s_1), \ldots, (l_n : s_n)$$

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

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

[6] All deviations we introduced are harmless.

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 [10]. 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

$$\mathbf{0} + r \Rightarrow r \qquad r + \mathbf{0} \Rightarrow r \qquad \mathbf{1} \cdot r \Rightarrow r \qquad r \cdot \mathbf{1} \Rightarrow r \tag{2}$$

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 [10] 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:

$$\begin{array}{lcl}
F_{Right}\, f\, v & \stackrel{\text{def}}{=} & Right\ (f\, v) \\
F_{Left}\, f\, v & \stackrel{\text{def}}{=} & Left\ (f\, v) \\
F_{Alt}\, f_1\, f_2\ (Right\ v) & \stackrel{\text{def}}{=} & Right\ (f_2\ v) \\
F_{Alt}\, f_1\, f_2\ (Left\ v) & \stackrel{\text{def}}{=} & Left\ (f_1\ v) \\
F_{Seq1}\, f_1\, f_2\ v & \stackrel{\text{def}}{=} & Seq\ (f_1\ ())\ (f_2\ v) \\
F_{Seq2}\, f_1\, f_2\ v & \stackrel{\text{def}}{=} & Seq\ (f_1\ v)\ (f_2\ ()) \\
F_{Seq}\, f_1\, f_2\ (Seq\ v_1\ v_2) & \stackrel{\text{def}}{=} & Seq\ (f_1\ v_1)\ (f_2\ v_2) \\
\\
simp_{Alt}\ (\mathbf{0}, _)\ (r_2, f_2) & \stackrel{\text{def}}{=} & (r_2, F_{Right}\, f_2) \\
simp_{Alt}\ (r_1, f_1)\ (\mathbf{0}, _) & \stackrel{\text{def}}{=} & (r_1, F_{Left}\, f_1) \\
simp_{Alt}\ (r_1, f_1)\ (r_2, f_2) & \stackrel{\text{def}}{=} & (r_1 + r_2, F_{Alt}\, f_1\, f_2) \\
simp_{Seq}\ (\mathbf{1}, f_1)\ (r_2, f_2) & \stackrel{\text{def}}{=} & (r_2, F_{Seq1}\, f_1\, f_2) \\
simp_{Seq}\ (r_1, f_1)\ (\mathbf{1}, f_2) & \stackrel{\text{def}}{=} & (r_1, F_{Seq2}\, f_1\, f_2) \\
simp_{Seq}\ (r_1, f_1)\ (r_2, f_2) & \stackrel{\text{def}}{=} & (r_1 \cdot r_2, F_{Seq}\, f_1\, f_2)
\end{array}$$

The functions $simp_{Alt}$ and $simp_{Seq}$ encode the simplification rules in (2) and compose the rectification functions (simplifications can occur deep inside the regular expression). 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 un-

der stars: this seems to slow down the algorithm, rather than speed it up. The optimised lexer is then given by the clauses:

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

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 4.
(1) $L(\mathit{fst}\ (\mathit{simp}\ r)) = L(r)$
(2) If $(s, \mathit{fst}\ (\mathit{simp}\ r)) \to v$ *then* $(s, r) \to \mathit{snd}\ (\mathit{simp}\ r)\ v$.

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 four subcases whether $\mathit{fst}\ (\mathit{simp}\ r_1)$ and $\mathit{fst}\ (\mathit{simp}\ r_2)$ equals $\mathbf{0}$ (respectively $\mathbf{1}$). For example for $r = r_1 + r_2$, consider the subcase $\mathit{fst}\ (\mathit{simp}\ r_1) = \mathbf{0}$ and $\mathit{fst}\ (\mathit{simp}\ r_2) \neq \mathbf{0}$. By assumption we know $(s, \mathit{fst}\ (\mathit{simp}\ (r_1 + r_2))) \to v$. From this we can infer $(s, \mathit{fst}\ (\mathit{simp}\ r_2)) \to v$ and by IH also (*) $(s, r_2) \to \mathit{snd}\ (\mathit{simp}\ r_2)\ v$. Given $\mathit{fst}\ (\mathit{simp}\ r_1) = \mathbf{0}$ we know $L(\mathit{fst}\ (\mathit{simp}\ r_1)) = \varnothing$. 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 $(s, r_1 + r_2) \to \mathit{Right}\ (\mathit{snd}\ (\mathit{simp}\ r_2)\ v)$. In turn this gives $(s, r_1 + r_2) \to \mathit{snd}\ (\mathit{simp}\ (r_1 + r_2))\ v$ 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 3. $\mathit{lexer}^+\ r\ s = \mathit{lexer}\ r\ s$

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 Thm. 1(2) a value v so that $\mathit{lexer}\ (r\backslash c)\ s = \mathit{Some}\ v$ and $(s, r\backslash c) \to v$ hold. By Lem. 4(1) we can also infer from (*) that $s \in L(r_s)$ holds. Hence we know by Thm. 1(2) that there exists a v' with $\mathit{lexer}\ r_s\ s = \mathit{Some}\ v'$ and $(s, r_s) \to v'$. From the latter we know by Lem. 4(2) that $(s, r\backslash c) \to f_r\ v'$ holds. By the uniqueness of the POSIX relation (Thm. 1) 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 $lexer\ (r\backslash c)\ s = None$ by Thm. 1(1). We also know by Lem. 4(1) that $s \notin L(r_s)$. Hence $lexer\ r_s\ s = None$ by Thm. 1(1) and by IH then also $lexer^+\ r_s\ s = None$. With this we can conclude in this case too. □

5 The Correctness Argument by Sulzmann and Lu

An extended version of [10] is available at the website of its first author; this includes some "proofs", claimed in [10] to be "rigorous". Since these are evidently not in final form, we make no comment thereon, preferring to give general reasons for our belief that the approach of [10] is problematic. Their central definition is an "ordering relation" defined by the rules (slightly adapted to fit our notation):

$$\frac{v_1 >_{r_1} v_1'}{Seq\ v_1\ v_2 >_{r_1 \cdot r_2} Seq\ v_1'\ v_2'}\ (C2) \qquad \frac{v_2 >_{r_2} v_2'}{Seq\ v_1\ v_2 >_{r_1 \cdot r_2} Seq\ v_1\ v_2'}\ (C1)$$

$$\frac{len\ |v_1| < len\ |v_2|}{Right\ v_2 >_{r_1 + r_2} Left\ v_1}\ (A1) \qquad \frac{len\ |v_2| \leq len\ |v_1|}{Left\ v_1 >_{r_1 + r_2} Right\ v_2}\ (A2)$$

$$\frac{v_1 >_{r_2} v_2}{Right\ v_1 >_{r_1 + r_2} Right\ v_2}\ (A3) \qquad \frac{v_1 >_{r_1} v_2}{Left\ v_1 >_{r_1 + r_2} Left\ v_2}\ (A4)$$

$$\frac{|Stars\ (v :: vs)| = []}{Stars\ [] >_{r^\star} Stars\ (v :: vs)}\ (K1) \qquad \frac{|Stars\ (v :: vs)| \neq []}{Stars\ (v :: vs) >_{r^\star} Stars\ []}\ (K2)$$

$$\frac{v_1 >_r v_2}{Stars\ (v_1 :: vs_1) >_{r^\star} Stars\ (v_2 :: vs_2)}\ (K3) \qquad \frac{Stars\ vs_1 >_{r^\star} Stars\ vs_2}{Stars\ (v :: vs_1) >_{r^\star} Stars\ (v :: vs_2)}\ (K4)$$

The idea behind the rules (A1) and (A2), for example, is that a *Left*-value is bigger than a *Right*-value, if the underlying string of the *Left*-value is longer or of equal length to the underlying string of the *Right*-value. The order is reversed, however, if the *Right*-value can match a longer string than a *Left*-value. In this way the POSIX value is supposed to be the biggest value for a given string and regular expression.

Sulzmann and Lu explicitly refer to the paper [3] by Frisch and Cardelli from where they have taken the idea for their correctness proof. Frisch and Cardelli introduced a similar ordering for GREEDY matching and they showed that their GREEDY matching algorithm always produces a maximal element according to this ordering (from all possible solutions). The only difference between their GREEDY ordering and the "ordering" by Sulzmann and Lu is that GREEDY always prefers a *Left*-value over a *Right*-value, no matter what the underlying string is. This seems to be only a very minor difference, but it has drastic consequences in terms of what properties both orderings enjoy. What is interesting for our purposes is that the properties reflexivity, totality and transitivity for this GREEDY ordering can be proved relatively easily by induction.

These properties of GREEDY, however, do not transfer to the POSIX "ordering" by Sulzmann and Lu, which they define as $v_1 \geq_r v_2$. To start with, $v_1 \geq_r v_2$ is not defined

inductively, but as $(v_1 = v_2) \vee (v_1 >_r v_2 \wedge |v_1| = |v_2|)$. This means that $v_1 >_r v_2$ does not necessarily imply $v_1 \geq_r v_2$. Moreover, transitivity does not hold in the "usual" formulation, for example:

Falsehood 1 *Suppose $v_1 : r$, $v_2 : r$ and $v_3 : r$. If $v_1 >_r v_2$ and $v_2 >_r v_3$ then $v_1 >_r v_3$.*

If formulated in this way, then there are various counter examples: For example let r be $a + ((a + a)\cdot(a + \mathbf{0}))$ then the v_1, v_2 and v_3 below are values of r:

$$\begin{array}{l} v_1 = \mathit{Left}\ (\mathit{Char}\ a) \\ v_2 = \mathit{Right}\ (\mathit{Seq}\ (\mathit{Left}\ (\mathit{Char}\ a))\ (\mathit{Right}\ ())) \\ v_3 = \mathit{Right}\ (\mathit{Seq}\ (\mathit{Right}\ (\mathit{Char}\ a))\ (\mathit{Left}\ (\mathit{Char}\ a))) \end{array}$$

Moreover $v_1 >_r v_2$ and $v_2 >_r v_3$, but *not* $v_1 >_r v_3$! The reason is that although v_3 is a *Right*-value, it can match a longer string, namely $|v_3| = [a, a]$, while $|v_1|$ (and $|v_2|$) matches only $[a]$. So transitivity in this formulation does not hold—in this example actually $v_3 >_r v_1$!

Sulzmann and Lu "fix" this problem by weakening the transitivity property. They require in addition that the underlying strings are of the same length. This excludes the counter example above and any counter-example we were able to find (by hand and by machine). Thus the transitivity lemma should be formulated as:

Conjecture 1 *Suppose $v_1 : r$, $v_2 : r$ and $v_3 : r$, and also $|v_1| = |v_2| = |v_3|$.*
If $v_1 >_r v_2$ and $v_2 >_r v_3$ then $v_1 >_r v_3$.

While we agree with Sulzmann and Lu that this property probably(!) holds, proving it seems not so straightforward: although one begins with the assumption that the values have the same flattening, this cannot be maintained as one descends into the induction. This is a problem that occurs in a number of places in the proofs by Sulzmann and Lu.

Although they do not give an explicit proof of the transitivity property, they give a closely related property about the existence of maximal elements. They state that this can be verified by an induction on r. We disagree with this as we shall show next in case of transitivity. The case where the reasoning breaks down is the sequence case, say $r_1 \cdot r_2$. The induction hypotheses in this case are

$$\begin{array}{ll} \text{IH } r_1\text{:} & \text{IH } r_2\text{:} \\ \forall\ v_1, v_2, v_3. & \forall\ v_1, v_2, v_3. \\ \quad v_1 : r_1 \wedge v_2 : r_1 \wedge v_3 : r_1 & \quad v_1 : r_2 \wedge v_2 : r_2 \wedge v_3 : r_2 \\ \quad \wedge\ |v_1| = |v_2| = |v_3| & \quad \wedge\ |v_1| = |v_2| = |v_3| \\ \quad \wedge\ v_1 >_{r_1} v_2 \wedge v_2 >_{r_1} v_3 & \quad \wedge\ v_1 >_{r_2} v_2 \wedge v_2 >_{r_2} v_3 \\ \quad \Rightarrow v_1 >_{r_1} v_3 & \quad \Rightarrow v_1 >_{r_2} v_3 \end{array}$$

We can assume that

$$\mathit{Seq}\ v_{1l}\ v_{1r} >_{r_1 \cdot r_2} \mathit{Seq}\ v_{2l}\ v_{2r} \qquad \text{and} \qquad \mathit{Seq}\ v_{2l}\ v_{2r} >_{r_1 \cdot r_2} \mathit{Seq}\ v_{3l}\ v_{3r} \tag{3}$$

hold, and furthermore that the values have equal length, namely:

$$|\mathit{Seq}\ v_{1l}\ v_{1r}| = |\mathit{Seq}\ v_{2l}\ v_{2r}| \qquad \text{and} \qquad |\mathit{Seq}\ v_{2l}\ v_{2r}| = |\mathit{Seq}\ v_{3l}\ v_{3r}| \tag{4}$$

We need to show that $Seq\ v_{1l}\ v_{1r} >_{r_1 \cdot r_2} Seq\ v_{3l}\ v_{3r}$ holds. We can proceed by analysing how the assumptions in (3) have arisen. There are four cases. Let us assume we are in the case where we know

$$v_{1l} >_{r_1} v_{2l} \qquad \text{and} \qquad v_{2l} >_{r_1} v_{3l}$$

and also know the corresponding inhabitation judgements. This is exactly a case where we would like to apply the induction hypothesis IH r_1. But we cannot! We still need to show that $|v_{1l}| = |v_{2l}|$ and $|v_{2l}| = |v_{3l}|$. We know from (4) that the lengths of the sequence values are equal, but from this we cannot infer anything about the lengths of the component values. Indeed in general they will be unequal, that is

$$|v_{1l}| \neq |v_{2l}| \qquad \text{and} \qquad |v_{1r}| \neq |v_{2r}|$$

but still (4) will hold. Now we are stuck, since the IH does not apply. As said, this problem where the induction hypothesis does not apply arises in several places in the proof of Sulzmann and Lu, not just for proving transitivity.

6 Conclusion

We have implemented the POSIX value calculation algorithm introduced by Sulzmann and Lu [10]. 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 [12]) appears to be much simpler than in [10] and our proofs are nearly always straightforward. We have attempted to formalise the original proof by Sulzmann and Lu [10], but we believe it contains unfillable gaps. In the online version of [10], the authors already acknowledge some small problems, but our experience suggests that there are more serious problems.

Having proved the correctness of the POSIX lexing algorithm in [10], 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 [10] appeared, but have made little progress with turning the relatively detailed proof sketch in [10] into a formalisable proof. Having seen [12] 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 [10], potentially using our result as a stepping stone, can be made to work? Alas, we really do not know despite considerable effort.

Closely related to our work is an automata-based lexer formalised by Nipkow [7]. 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 lexer for the Imp-language is formalised in Coq as part of the Software Foundations book by Pierce et al [9]. The disadvantage of such bespoke lexers is that they do not generalise easily to more advanced features. Our formalisation is available from http://www.inf.kcl.ac.uk/staff/urbanc/lex.

Acknowledgements: We are very grateful to Martin Sulzmann for his comments on our work and moreover patiently explaining to us the details in [10]. We also received very helpful comments from James Cheney and anonymous referees.

References

1. J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.
2. 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.
3. A. Frisch and L. Cardelli. Greedy Regular Expression Matching. In *Proc. of the 31st International Conference on Automata, Languages and Programming (ICALP)*, volume 3142 of *LNCS*, pages 618–629, 2004.
4. N. B. B. Grathwohl, F. Henglein, and U. T. Rasmussen. A Crash-Course in Regular Expression Parsing and Regular Expressions as Types. Technical report, University of Copenhagen, 2014.
5. A. Krauss and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *Journal of Automated Reasoning*, 49:95–106, 2012.
6. C. Kuklewicz. Regex Posix. https://wiki.haskell.org/Regex_Posix.
7. T. Nipkow. Verified Lexical Analysis. In *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1479 of *LNCS*, pages 1–15, 1998.
8. S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.
9. B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2015. http://www.cis.upenn.edu/~bcpierce/sf.
10. M. Sulzmann and K. Lu. POSIX Regular Expression Parsing with Derivatives. In *Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS)*, volume 8475 of *LNCS*, pages 203–220, 2014.
11. M. Sulzmann and P. van Steenhoven. A Flexible and Efficient ML Lexer Tool Based on Extended Regular Expression Submatching. In *Proc. of the 23rd International Conference on Compiler Construction (CC)*, volume 8409 of *LNCS*, pages 174–191, 2014.
12. S. Vansummeren. Type Inference for Unique Pattern Matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.