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 it seems hard to turn it into a proof rigorous enough to be accepted by a system such as Isabelle/HOL.

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

completely formalised correctness proof of this matcher in for example HOL4 has been mentioned in [5]. Another one in Isabelle/HOL is part of the work in [3].

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 [6] 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 [4,7]. 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 [6] 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 [2] on a GREEDY regular expression matching algorithm. Beginning with our observations that, without evidence that it is transitive, it cannot be called an "order relation", and that the relation is called a "total order" despite being evidently not total[4], we identify problems with this approach (of which some of the proofs are not published in [6]); 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. 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 Kuklewitz [4] who found that nearly all POSIX matching implementations are "buggy" [6, Page 203].

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 [2] and the other is POSIX matching [4,6,7]. 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

[4] The relation $\geq r$ defined in [6] is a relation on the values for the regular expression r; but it only holds between v and v' in cases where v and v' have the same flattening (underlying string). So a counterexample to totality is given by taking two values v and v' for r that have different flattenings (see Section 3). A different relation $\geq_{r,s}$ on the set of values for r with flattening s is definable by the same approach, and is indeed total; but that is not what Proposition 1 of [6] does.

identifiers, respectively. There are two underlying (informal) rules behind tokenising a string in a POSIX 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 regular expression that can match determines the token.

Consider for example r_{key} 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.

Contributions: (NOT DONE YET) We have implemented in Isabelle/HOL the derivative-based regular expression matching algorithm as described by Sulzmann and Lu [6]. We have proved the correctness of this algorithm according to our specification of what a POSIX value is. Sulzmann and Lu sketch in [6] an informal correctness proof: but to us it contains unfillable gaps.

informal correctness proof given in [6] is in final form[5] and to us contains unfillable gaps.

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, such as the simplifications of $\mathbf{0} + r$, $r + \mathbf{0}$, $\mathbf{1} \cdot r$ and $r \cdot \mathbf{1}$ to r. One of the advantages of having a simple specification and correctness proof is that the latter can be refined to allow for such optimisations and simple correctness proof.

An extended version of [6] is available at the website of its first author; this includes some "proofs", claimed in [6] 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 [6] is problematic rather than to discuss details of unpublished work.

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:

[5]

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

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 clauses:

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

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:

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

For semantic derivatives we have the following equations (for example mechanically proved in [3]):

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

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) $L(r\backslash c) = \mathit{Der}\ c\ (L(r))$.

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 [6] 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 in [6] is to introduce 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

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

where we use *vs* to stand for a list of values. (This is similar to the approach taken by Frisch and Cardelli for GREEDY matching [2], and Sulzmann and Lu [6] for POSIX matching). The string underlying a value can be calculated by the *flat* function, written $|\_|$ and defined as:

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

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

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

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

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

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

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

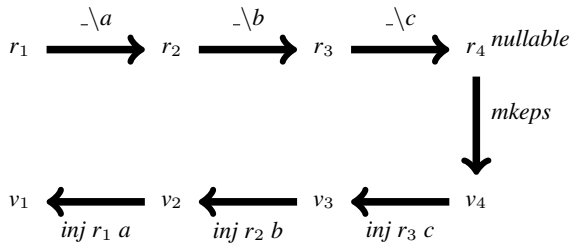

Fig. 1. The two phases of the algorithm by Sulzmann & Lu [6], matching the string $[a, b, c]$. The first phase (the arrows from left to right) is Brzozowski's matcher building succesive derivatives. If the last regular expression is *nullable*, then the functions of the second phase are called (the top-down and right-to-left arrows): first *mkeps* calculates a value 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.

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 rules in case there are several ways). This functions 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 an error is raised instead. 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 [6] 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 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 corresponding to the input regular expression. The *inj* function is by recursion on the 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:

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

Consider first the else-branch where the derivative is $(r_1\backslash c) \cdot r_2$. The corresponding value must therefore be 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 $r \cdot r^\star$. 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 is to build this constraint explicitly into our function definition.[6]

The idea of *inj* to "inject back" a character into a value can be made precise by the first part of the following lemma; the second part shows that the underlying string of an *mkeps*-value is always the empty string.

Lemma 1.
(1) If $v : r\backslash c$ *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, for example, by an induction over the definition of *derivatives*; the second by 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 lexer are

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

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

If the regular expression does not match the string, *None* is returned, indicating an error is raised. 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 a 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; as correctly argued in [6], this needs formal specification. Sulzmann and Lu define a *dominance* relation[7] 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 in [7]. The relation we define is ternary and written as $(s, r) \to v$, relating strings, regular expressions and values.

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

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

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

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

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

We claim that this 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 an alternative regular expression 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

[7] Sulzmann and Lu call it an ordering relation, but without giving evidence that it is transitive.

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 *exists* an s_3 and s_4 such that s_2 can be split up into a non-empty s_3 and s_4. Moreover the longer $s_1 @ s_3$ 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 $Seq\ v_1\ v_2$ cannot be a POSIX value for $(s_1 @ s_2, r_1 \cdot r_2)$. A similar condition is imposed onto the POSIX value in the $P\star$-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 flatten to the empty string. In effect, we require that in each "iteration" of the star, some parts of the string need to be "nibbled" away; only in case of the empty string weBy accept *Stars* $[]$ as the POSIX value.

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. *If* $(s, r) \to v_1$ *and* $(s, r) \to v_2$ *then* $v_1 = v_2$.

Proof. By induction on the definition of $(s, r) \to v_1$ and a case analysis of $(s, r) \to v_2$. □

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. Suppose $r = r_1 + r_2$. 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 $s \notin L(r_1\backslash c)$.

Suppose $r = r_1 \cdot r_2$. 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

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

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

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

We can use the induction hypothesis for r_1 to obtain $(c :: s_1, r_1) \to inj\ r_1\ c\ v_1$. This allows us to infer $(c :: s_1 @ s_2, r_1 \cdot r_2) \to Seq\ (inj\ r_1\ c\ v_1)\ v_2$. The case (c) is similarly.

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 $(c :: s, r_2) \to \mathit{inj}\ r_2\ c\ v_1$ 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 $([], r_1) \to \mathit{mkeps}\ r_1$ holds. Putting this all together, we can conclude with $(c :: s, r_1 \cdot r_2) \to \mathit{Seq}\ (\mathit{mkeps}\ r_1)\ (\mathit{inj}\ r_2\ c\ v_1)$.

Finally suppose $r = r_1{}^\star$. This case is very similar to the sequence case, except that we need to ensure that $|\mathit{inj}\ r_1\ c\ v_1| \neq []$. This follows by Lem. **??** from $(c :: s_1, r') \to \mathit{inj}\ r_1\ c\ v_1$ (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 its specification (returning an "error" 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 $\wedge$ $(s, r) \to v$

Proof. By induction on s. □

This concludes our correctness proof. Note that we have not changed the algorithm by Sulzmann and Lu, but introduced our own specification for what a correct result—a POSIX value—should be.

4 The Argument by Sulzmmann and Lu

5 Conclusion

Nipkow lexer from 2000

We have also introduced a slightly restricted version of this relation where the last rule is restricted so that $|v| \neq []$.

Our version of Sulzmann's ordering relation

Some lemmas we have proved:

$L(r) = \{|v| \mid v : r\}$
If nullable r then mkeps r : r.
If nullable r then $|\mathit{mkeps}\ r| = []$.
If $v : r\backslash c$ then inj r c v : r.
If $v : r\backslash c$ then $|\mathit{inj}\ r\ c\ v| = c :: |v|$.
If nullable r then $([], r) \to \mathit{mkeps}\ r$.
If $(s, r) \to v$ then $|v| = s$.

If $(s, r) \to v_1$ *and* $(s, r) \to v_2$ *then* $v_1 = v_2$.

Proof The proof is by induction on the definition of *der*. Other inductions would go through as well. The interesting case is for $r_1 \cdot r_2$. First we analyse the case where *nullable* r_1. We have by induction hypothesis

$$\begin{array}{ll}(IH1) & \forall s\ v.\ \text{if } (s, r_1\backslash c) \to v \text{ then } (c :: s, r_1) \to \mathit{inj}\ r_1\ c\ v\\ (IH2) & \forall s\ v.\ \text{if } (s, r_2\backslash c) \to v \text{ then } (c :: s, r_2) \to \mathit{inj}\ r_2\ c\ v\end{array}$$

and have

$$(s, (r_1\backslash c) \cdot r_2 + (r_2\backslash c)) \to v$$

There are two cases what v can be: (1) *Left* v' and (2) *Right* v'.

(1) We know $(s, (r_1\backslash c) \cdot r_2) \to v'$ holds, from which we can infer that there are s_1, s_2, v_1, v_2 with

$$(s_1, r_1\backslash c) \to v_1 \qquad \text{and} \qquad (s_2, r_2) \to v_2$$

and also

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

and have to prove

$$(c :: s_1 \mathbin{@} s_2, r_1 \cdot r_2) \to \mathit{Seq}\ (\mathit{inj}\ r_1\ c\ v_1)\ v_2$$

The two requirements $(c :: s_1, r_1) \to \mathit{inj}\ r_1\ c\ v_1$ and $(s_2, r_2) \to v_2$ can be proved by the induction hypothesis (IH1) and the fact above.
This leaves to prove

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

which holds because $c :: s_1 \mathbin{@} s_3 \in L(r_1)$ implies $s_1 \mathbin{@} s_3 \in L(r_1\backslash c)$

(2) This case is similar.

The final case is that $\neg$ *nullable* r_1 holds. This case again similar to the cases above.

References

1. J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.
2. 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.
3. A. Krauss and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *Journal of Automated Reasoning*, 49:95–106, 2012.
4. C. Kuklewicz. Regex Posix. https://wiki.haskell.org/Regex_Posix.
5. S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.

6. 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.
7. S. Vansummeren. Type Inference for Unique Pattern Matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.