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 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 given in [4].

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 [5] 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 lexing [3,6]. 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 [5] 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 [5]); 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 [3] who found that nearly all POSIX matching implementations are "buggy" [5, 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 [3,5]. 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:

- The Longest Match Rule (or "maximal munch rule"):
  The longest initial substring matched by any regular expression is taken as next token.

[4] We should give an argument as footnote

- Rule Priority:
  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 (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*. In the first case we obtain by the longest match rule a single identifier token, not a keyword followed by identifier. In the second case we obtain by rule priority a keyword token, not an identifier token—even if r_{id} matches also.

Not Done Yet

Contributions:

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 [5] is available at the website of its first author; this includes some "proofs", claimed in [5] 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 [5] 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 strings; for example a string consisting of just a single character 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 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 and $\mathbf{1}$ for the regular expression that matches only the empty string. The language of a regular expression is again defined routinely by the recursive function L with the clauses:

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

In the fourth clause we use $\_ \mathbin{@} \_$ for the concatenation of two languages (it is also list-append for strings). We use the star-notation for regular expressions and also for languages (in the last clause). The star for languages is defined inductively as usual by two clauses for the empty string being in the star of a language and 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.

Semantic derivatives of sets of strings are defined as

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

The nullable function

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

The derivative function for characters and strings

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

It is a relatively easy exercise to prove that

$$\begin{array}{l}
nullable\ r = ([] \in L(r)) \\
L(r\backslash c) = Der\ c\ (L(r))
\end{array}$$

3 POSIX Regular Expression Matching

The clever idea in [5] is to define a function on values that mirrors (but inverts) the construction of the derivative on regular expressions. We begin with the case of a nullable regular expression: from the nullability we need to construct a value that witnesses the nullability. This is as follows. The *mkeps* function (from [5]) is a partial (but unambiguous) function from regular expressions to values, total on exactly the set of nullable regular expressions.

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

The well-known idea of POSIX lexing is informally defined in (for example) [**?**]; as correctly argued in [5], this needs formal specification. The rough idea is that, in contrast to the so-called GREEDY algorithm, POSIX lexing chooses to match more deeply and using left choices rather than a right choices. For example, note that to match the string $[a, b]$ with the regular expression $(a + \varepsilon) \circ (b + ab)$ the matching will return $(Void, Right(ab))$ rather than $(Left\ a, Left\ b)$. [The regular expression ab is short for $(Lit\ a) \circ (Lit\ b)$.] Similarly, to match *"a"* with $(a + a)$ the leftmost a will be chosen.

We use a simple inductive definition to specify this notion, incorporating the POSIX-specific choices into the side-conditions for the rules $Rtl{+_2}$, $Rtl{\circ}$ and $Rtl{*}$ (as they are now called). By contrast, [5] defines a relation between values and argues that there is a maximum value, as given by the derivative-based algorithm yet to be spelt out. The relation we define is ternary, relating strings, values and regular expressions.

Our Posix relation $s \in r \to v$

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

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

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

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

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

4 The Argument by Sulzmmann and Lu

5 Conclusion

Nipkow lexer from 2000

Values

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

The *flat* function for values

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

The *mkeps* function
The *inj* function

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

The inhabitation relation:

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

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

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

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

We have also introduced a slightly restricted version of this relation where the last rule is restricted so that $|v| \neq []$. This relation for *non-problematic* is written $\models v : r$.

Our version of Sulzmann's ordering relation

$$\frac{v_1 \succ_{r_1} v_1' \qquad v_1 \neq v_1'}{Seq\ v_1\ v_2 \succ_{(r_1 \cdot r_2)} Seq\ v_1'\ v_2'} \qquad \frac{v_2 \succ_{r_2} v_2'}{Seq\ v_1\ v_2 \succ_{(r_1 \cdot r_2)} Seq\ v_1\ v_2'}$$

$$\frac{len\ (|v_1|) \leq len\ (|v_2|)}{(Left\ v_2) \succ_{(r_1 + r_2)} (Right\ v_1)} \qquad \frac{len\ (|v_2|) < len\ (|v_1|)}{(Right\ v_1) \succ_{(r_1 + r_2)} (Left\ v_2)}$$

$$\frac{v_2 \succ_{r_2} v_2'}{(Right\ v_2) \succ_{(r_1 + r_2)} (Right\ v_2')} \qquad \frac{v_1 \succ_{r_1} v_1'}{(Left\ v_1) \succ_{(r_1 + r_2)} (Left\ v_1')}$$

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

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

$$\frac{v_1 \succ_r v_2 \qquad v_1 \neq v_2}{Stars\ (v_1 :: vs_1) \succ_{(r^\star)} Stars\ (v_2 :: vs_2)}$$

$$\frac{Stars\ vs_1 \succ_{(r^\star)} Stars\ vs_2}{Stars\ (v :: vs_1) \succ_{(r^\star)} Stars\ (v :: vs_2)} \qquad \frac{}{Stars\ [] \succ_{(r^\star)} Stars\ []}$$

A prefix of a string s

$$s_1 \sqsubseteq s_2 \stackrel{def}{=} \exists s3.\ s_1 @ s3 = s_2$$

Values and non-problematic values

$$Values\ r\ s \stackrel{def}{=} \{v \mid \vdash v : r \wedge (|v|) \sqsubseteq s\}$$

The point is that for a given s and r there are only finitely many non-problematic values.

Some lemmas we have proved:

$L(r) = \{|v| \mid \vdash v : r\}$
$L(r) = \{|v| \mid \models v : r\}$
If $nullable\ r$ *then* $\vdash mkeps\ r : r$.
If $nullable\ r$ *then* $|mkeps\ r| = []$.
If $\vdash v : (r\backslash c)$ *then* $\vdash (inj\ r\ c\ v) : r$.
If $\vdash v : (r\backslash c)$ *then* $|inj\ r\ c\ v| = c :: (|v|)$.
If $nullable\ r$ *then* $[] \in r \rightarrow mkeps\ r$.
If $s \in r \rightarrow v$ *then* $|v| = s$.
If $s \in r \rightarrow v$ *then* $\models v : r$.
If $s \in r \rightarrow v_1$ *and* $s \in r \rightarrow v_2$ *then* $v_1 = v_2$.

This is the main theorem that lets us prove that the algorithm is correct according to $s \in r \rightarrow v$:
If $s \in (r\backslash c) \rightarrow v$ *then* $(c :: s) \in r \rightarrow (inj\ r\ c\ v)$.

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

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

and have

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

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

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

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

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

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

The two requirements $(c :: s_1) \in r_1 \to (\mathit{inj}\ r_1\ c\ v_1)$ and $s_2 \in r_2 \to v_2$ can be proved by the induction hypothese (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. C. Kuklewicz. Regex Posix. https://wiki.haskell.org/Regex_Posix.
4. S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.
5. 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.
6. S. Vansummeren. Type Inference for Unique Pattern Matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.

6 Roy's Rules

$$Void \lhd \epsilon \qquad Char\ c \lhd Lit\ c$$

$$\frac{v_1 \lhd r_1}{Left\ v_1 \lhd r_1 + r_2} \qquad \frac{v_2 \lhd r_2 \qquad |v_2| \notin L(r_1)}{Right\ v_2 \lhd r_1 + r_2}$$

$$\frac{v_1 \lhd r_1 \qquad v_2 \lhd r_2 \qquad s \in L(r_1 \backslash |v_1|) \wedge |v_2| \backslash s \notin L(r_2) \Rightarrow s = []}{(v_1, v_2) \lhd r_1 \cdot r_2}$$

$$\frac{v \lhd r \qquad vs \lhd r^* \qquad |v| \neq []}{(v :: vs) \lhd r^*} \qquad [] \lhd r^*$$