Matching problem for regular expressions with variables

Extended abstract

Vladimir Komendantsky

University of St Andrews, UK
http://www.cs.st-andrews.ac.uk/∼vk/

Abstract. We study the notion of *backreference* in practical regular expressions with the purpose of formal analysis and theorem proving. So far only their operational semantics was studied in formal language theory. However, for efficient reasoning and independence from implementations we need a mathematical concept of backreference that does not depend on evaluation strategies, etc. We introduce such a notion in terms of a possibly infinite set of finite tree unfoldings of regular expressions for which we state the finiteness of a partial derivative (actually, prebase) representation.

1 Introduction

Regular expressions [14, 10] are a formalism ideally suited to specification and implementation with formal methods. They are essential for text processing and form the basis of most markup schema languages. Regular expressions are useful in the production of syntax highlighting systems, data validation, speech processing, optical character recognition, and in many other situations when we attempt to recognise patterns in data.

Extended versions of regular expressions are used in search engines such as Google Code Search. In fact, there is a difference between what is understood by the term *regular expression* in programming and in theoretical computer science. Different software based on regular expressions has in each case its own "RegEx flavour": ECMAScript, Perl-style, GNU RegEx, Microsoft Word, POSIX Basic/Extended RegEx (with extensions), Vim, and many others.

In contrast, theoretical computer science uses a single formal definition for a regular expression which defines it as consisting of constants and operators that denote sets of strings and operations over these sets respectively. For example, assuming a and b are symbols, $\underline{a}+\underline{b}^*$ denotes the set $\{\epsilon, a, b, bb, bbb, \dots\}$, where ϵ denotes the empty string; and $(\underline{a}+\underline{b})^*$ denotes the set of all strings composed of a and b (including the empty string ϵ): $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$. The formal definition is purposely minimalist in that it avoids redundant operators that can be expressed through application of existing ones. Regular expressions in the precise sense express the class of *regular languages*, which is exactly the class of languages accepted by *finite state automata.* This definition has the maximum degree of independence from any implementation.

In functional programming, the usual procedure is to enumerate words of the languages denoted by regular expressions using finite automata [11]. Less common but more efficient are partial derivatives of regular expressions [12, 3]. They are purely functional [9].

Patterns. A *pattern* [6] is a description of a word at a meta-scale: instead of considering the word as a sequence of individual symbols, one looks at the word as a sequence of certain blocks. More formally, a pattern p is a word that contains special symbols, called *pattern variables*; p is a pattern of a word w if w is obtained from p by uniformly replacing the variables with words (which may be empty depending on the convention). The *pattern language* denoted by a pattern p is the set of all words that match p. For example, the pattern xyx denotes the set of all words in a language each of which has a prefix and a suffix that are the same and have a word in between.

Many features found in modern pattern matching libraries, and in languages such as Perl or Python, provide expressive power that far exceeds the capacities of regular languages and requires pattern languages. In our opinion, the most interesting of such extra capacities is *backreferencing* – the ability to group subexpressions and recall the value they match later in the same expression. Such a pattern can match strings of repeated words like "papa", called squares in formal language theory. The obvious Perl pattern for such strings is `(.*)\1`. However, the language of squares is not regular, nor is it context-free. Regular expression matching with an unbounded number of backreferences, as supported by numerous modern tools, is NP-complete [1], and therefore requires practical algorithms that maximally approximate non-deterministic complexity bounds.

Backreferences may be seen acting as variables in patterns. The language denoted by a pattern is obtained by substituting variables with arbitrary terminal strings in the case on erasing patterns and non-empty terminal strings in the case of non-erasing patterns.

Pattern languages are often accepted as the theoretical meaning of extended, also called *practical*, regular expressions [5, 13]. However, patterns are defined with no recursion or choice and therefore require pre-processing of regular expressions in order to form a set of patterns from an expression with iteration or non-deterministic choice. For that reason, in this particular paper, we do not choose patterns as the meaning of backreferences.

Matching problem. The matching problem for regular expressions is the problem to decide, for a given word over a finite alphabet of symbols and a given regular expression, whether or not the word belongs to the language denoted by the regular expression. There are quite efficient polynomial algorithms for deciding this basic problem [12, 3, 7, 9]. These algorithms are based on a simple fact that the non-deterministic finite automaton recognising the language of the regular expression is guaranteed to terminate on finite input words.

Alternative existing definitions of regular expressions with backreferences, for example, by means of match-trees [4] or ordered trees [5], are better than patterns in terms of correspondence to the implementation of practical regular expressions – since they refer to the operational matching semantics – but, on the other hand, the tree semantics is quite specific to their matching problem.

The solution to these odds that we propose in this paper is to combine the two approaches: patterns and trees. The downside of patterns can be overcome if we consider patterns over convenient tree unfoldings of regular expressions instead of patterns over plain strings. Pattern variables then become symbolic variables ranging over regular expressions.

Outline. In the paper we recall definitions of patterns and match-trees for regular expressions with backreferences, and give a combined treatment to provide a tree semantics for regular expressions with variables. We then extend the language of regular expressions with constraints that define the meaning of variables.

2 Pattern languages

Let Σ be a finite alphabet of terminal symbols and X an infinite set of variables such that $\Sigma \cap X = \emptyset$. A *pattern* is a non-empty string over $\Sigma \cup X$, a *terminal-free pattern* is a non-empty string over X, and a word is a string over Σ. The set of variables of a pattern α is denoted $\mathrm{Var}(\alpha)$.

For any alphabets A and B, a *morphism from A to B* is a function $h : A^* \to B^*$ that satisfies $h(u \cdot w) = h(u) \cdot h(w)$ for all $u, w \in A^*$. A morphism $\sigma : (\Sigma \cup X)^* \to \Sigma^*$ is a *substitution* if $\sigma(a) = a$ for every $a \in \Sigma$.

The *matching problem for patterns* is the problem to decide, given a pattern α and a word $w \in \Sigma^*$, whether $w \in L_\Sigma(\alpha)$.

The following automaton construction can be used to recognise pattern languages [13]. That is, we can construct an automaton M satisfying $L(M) = L_\Sigma(\alpha)$, for a terminal-free pattern α. A *Janus automaton* $\mathrm{JFA}(k)$ is a 2-way 2-head automaton with k bounded counters. It is a tuple $M = (K, Q, \Sigma, \delta, q_0, F)$ where K is a set of k bounded counters, Σ is an input alphabet, δ is a transition function, Q is a set of states, $F \subseteq Q$ is a set of final states, and $q_0 \in Q$ is the initial state. Each computation step of the automaton sets up a counter bound for each counter. The counters accept operations of increment by 1, noop, and reset with 0. In case of increment, the next counter value is computed modulo the bound. In case of reset, the new counter bound is non-deterministically guessed.

Reidenbach and Schmid [13] stated the following polynomial complexity result for pattern languages. Let $\mathrm{vd}(\alpha)$ denote the maximum number of variables separating any two consecutive occurrences of any variable in a pattern α.

Theorem 1. *For a terminal-free pattern α and $w \in \Sigma^*$, there is a* $\mathrm{JFA}(\mathrm{vd}(\alpha)+1)$ *that decides $w \in L_\Sigma(\alpha)$ in time $O(|\alpha|^3|w|^{(\mathrm{vd}(\alpha)+4)})$.*

The polynomial worst-case complexity is achieved at the expense of introducing a non-deterministic reset feature, among other things. Although we would like to be able to reason about such algorithms, it does not seem possible to implement Janus automata deterministically. Moreover, state management issues are excessive, which makes these automata intangible from type theory. We need a more functional and less procedural definition of matching semantics. Such a definition appears in terms of ordered trees and is outlined in the next section.

3 Language-theoretic definition of regular expressions with backreferences

Let us assume that the opening parentheses in a given formal expression E are numbered from the left to the right, and the closing parentheses are numbered in the correspondence with the opening parentheses. For example,

$$\underset{1}{(} \ldots \underset{2}{(} \ldots \underset{3}{(} \ldots \underset{3}{)} \ldots \underset{2}{)} \ldots \underset{4}{(} \ldots \underset{4}{)}\, \underset{1}{)}$$

Definition 1 (Backreference). *A* backreference $\backslash m$ *where* $m \geq 1$, *matches the contents of the* m*-th pair of numbered parentheses on the left of it.*

For example, the regular expression $(\underline{a}^*)\times\underline{b}\times\backslash 1$ defines the language

$$\{a^n \cdot b \cdot a^n \mid n \geq 0\}$$

Lemma 1 (CSY pumping lemma, [4]). *Let E be a rewb. Then there is a natural number n such that, for $w \in L(E)$ and $|w| > n$, there is an $m \geq 1$ and a decomposition $w = x_0 \cdot y \cdot x_1 \cdot y \cdot \ldots \cdot x_m$ such that*

1. $|x_0 \cdot y| \leq n$,
2. $|y| \geq 1$,
3. $x_0 \cdot y^j \cdot x_1 \cdot y^j \cdot \ldots \cdot x_m \in L(E)$, *for all* $j > 0$.

From [4] it is known that rewb languages are context-sensitive and incomparable with the family of context-free languages. The latter is due to the observation that the language $\{a^n \cdot b^n \mid n \geq 0\}$ is context-free but cannot be expressed by a rewb (as a corollary of Lemma 1), and the language $\{a^n \cdot b \cdot a^n \cdot b \cdot a^n \mid n \geq 1\}$ is a rewb language but not a context-free one.

In [5], it was proved that rewb languages are not closed under intersection and their emptiness of intersection problem is undecidable. Moreover, since [1] we know that *the matching problem for rewbs is NP-complete* for arbitrary alphabets, and the paper [5] proves that NP-completeness holds even when the target string is over a unary alphabet.

More specifically, to define a matching problem for rewbs we quote the definition of *matching of a string with an rewb* from [5]. It uses the notion of an ordered tree. An ordered tree T is a valid match-tree for w and E if and only if the root of T is labelled by (w, E) and the following conditions hold for every node $u \in Dom(T)$:

1. If $T(u) = (w, \underline{a})$ for some $a \in A$ then u is a leaf node and $w = a$.

2. If $T(u) = (w, F_1{\times}F_2)$ then u has two children labelled, respectively, by (w_1, F_1) and (w_2, F_2), with $w_1 \cdot w_2 = w$.
3. If $T(u) = (w, F_1 + F_2)$ then u has one child labelled either by (w, F_1) or by (w, F_2).
4. If $T(u) = (w, F^*)$ then either u is a leaf node and $w = \epsilon$ or u has $k \geq 1$ children labelled by $(w_1, F), \ldots, (w_k, F)$, with $w_1 \cdot w_k = w$.
5. If $T(u) = (w, \underset{i}{(}F\underset{i}{)})$ then it has one child labelled by (w, F).
6. If $T(u) = (w, \backslash m)$ then u is a leaf node, $\underset{m}{(}\,F\,\underset{m}{)}$ is a subexpression of E, and there is a node v to the left of u such that $T(v) = (w, \underset{m}{(}\,F\,\underset{m}{)})$ and no node between v and u has $\underset{m}{(}\,F\,\underset{m}{)}$ in its label. In other words, w is the string previously matched by $\underset{m}{(}\,F\,\underset{m}{)}$ in the left-to-right pre-order of nodes.

The paper [4] features a slightly different definition where unassigned backreferences are set to match the empty string by default.

The language $L(E)$ denoted by a rewb E is the set of all $w \in \Sigma^*$ such that (w, E) is the root label of a valid match-tree. Thus we can state the following problem.

Definition 2 (Matching problem for rewbs). *For some w and E, is (w, E) the root label of a valid match tree?*

Unlike the case with Janus automata, there is an explicit search tree structure, although there does not seem to be a specific computational machine dedicated to the task of proof search for the matching problem in the literature. Meanwhile proof search is the most interesting part here. In the next section we elaborate on the tree semantics of matching, completely separating regular expressions and the trees they denote.

4 Regular expressions with variables

We define *regular expressions with variables (revs)* using by induction as follows:

$$E ::= 0 \mid 1 \mid \underline{a} \mid x \mid E + E \mid E{\times}E \mid E^*$$

for any $a \in \Sigma$ and any $x \in X$.

In practical regular expressions, the variables (i.e., backreferences) are assigned values in regular expressions. Therefore we have to express this kind of assignments. We can do that for a rev E by introducing a telescopic environment Δ consisting of equalities $x_1 = E_1,\ \ldots,\ x_n = E_n$, where $x_1, \ldots, x_n \in \mathrm{Var}(E)$ are distinct variable names and x_i does not occur in E_k for all $k \leq i$. The latter is required to exclude circularity. We say that E is *well-defined by Δ* if $\mathrm{Var}(E) = \mathrm{Dom}(\Delta)$ and the ordering of variables in Δ coincides with the left-to-right ordering of the first occurrences of corresponding free variables in E.

For E well-defined by Δ, let us add one more clause in the definition of one-step unfolding:

$$!x = \Delta(x)$$

for $x \in \mathrm{Var}(E)$. This a template definition of unfolding that can be modified to allow to experiment with different version of operational matching semantics. For example, one has the straightforward matching with backpropagation as in [1] by modifying the definition of one-step unfolding to enumerate possible candidate matches for occurrences of the variable. We omit the definition in this short paper.

The *tree semantics* of a rev E well-defined by Δ is the potentially infinite set of all finite unfoldings of subterms of E. Thinking in terms of algebra, we arrive at a conceptually simpler definition compared to those outlined in the previous two sections. The *prebase set* $\pi_\Delta(E)$ of E is defined by induction as follows (in the style of [2], modified from [12]):

$$\begin{aligned}
&\pi_\Delta(0) = \emptyset && \pi_\Delta(F + G) = \pi_\Delta(F) \cup \pi_\Delta(G)\\
&\pi_\Delta(1) = \emptyset && \pi_\Delta(F \times G) = \pi_\Delta(F) \cdot G \cup \pi_\Delta(G)\\
&\pi_\Delta(\underline{a}) = \{\epsilon\} && \pi_\Delta(F^*) = \pi_\Delta(F) \cdot F^*\\
&\pi_\Delta(x) = \pi(!x) &&
\end{aligned}$$

Since Δ is non-circular, and all the variables in E are defined in Δ, the clause for μ above does not introduce divergence. Therefore we have the following theorem:

Theorem 2. *For E well-defined by Δ, the set $\pi_\Delta(E)$ is finite.*

We omit the formulation of the prebase transition matrix. That matrix and the set of states $\{E\} \cup \pi_\Delta(E)$ yield a non-deterministic finite automaton accepting the language of E. Remarkably, we have the following corollary extending the results on regular expressions [12, 3]:

Corollary 1. *A word $w \in \Sigma^*$ is contained in the language of E (well-defined by Δ) if and only if all proper suffixes of w are contained in $L(E) \cup L(\bigcup \pi_\Delta(E))$.*

Therefore the matching problem for revs is effectively decided by the prebase construction. The nice property of this construction is that we have to compute prebase only once for a given regular expression, and then matching a word against it does not require any further computations but only traversal using the transition matrix.

5 Conclusions

We discussed a template for formal semantics of backreferences in regular expressions. It can be implemented in a pure functional style, in particular, in constructive type theory. The obtained semantics in terms of prebases is purely denotational. Matching with backreferences can be implemented on top of it using backpropagation. Nevertheless that version of matching – using partial derivatives – is different from the standard backtracking semantics of backreferences and therefore deserves further study in terms of complexity and possible optimisations, such as workarounds to tame backpropagation. The current work in progress concerns a certified matching algorithm: We aim at proving on a computer that proof search of the matching problem terminates for any input word and any well-defined regular expression with variables.

References

1. A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 255–300. The MIT Press, 1990.
2. J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa. Partial derivative automata formalized in Coq. In *Implementation and Application of Automata 2010*, volume 6482/2011 of *Lecture Notes in Computer Science*, pages 59–68, 2011.
3. V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.
4. C. Campeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14:1007 – 1018, 2003.
5. B. Carle and P. Narendran. On extended regular expressions. In *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications*, LATA '09, pages 279–289, Berlin, Heidelberg, 2009. Springer-Verlag.
6. G. Castiglione, A. Restivo, and S. Salemi. Patterns in words and languages. *Discrete Appl. Math.*, 144(3):237–246, 2004.
7. J.-M. Champarnaud and D. Ziadi. Canonical derivatives, partial derivatives and finite automaton constructions. *Theor. Comput. Sci.*, 289(1):137–163, 2002.
8. V. Komendantsky. Formal proofs of the prebase theorem of Mirkin, 2011. Coq script available at http://www.cs.st-andrews.ac.uk/~vk/doc/prebase.v.
9. V. Komendantsky. Reflexive toolbox for regular expression matching: Verification of functional programs in Coq+Ssreflect. In *The 6th ACM SIGPLAN Workshop Programming Languages meet Program Verification (PLPV'12)*, Philadelphia, USA, 24 January 2012. For contributed proofs, see [8].
10. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, 1994.
11. M. D. McIlroy. Enumerating the strings of regular languages. *Journal of Functional Programming*, 14:503–518, 2004.
12. B. G. Mirkin. New algorithm for construction of base in the language of regular expressions. *Tekhnicheskaya Kibernetika*, 5:113–119, 1966. English translation in *Engineering Cybernetics*, No. 5, Sept.–Oct. 1966, pp. 110-116.
13. D. Reidenbach and M. Schmid. A polynomial time match test for large classes of extended regular expressions. In M. Domaratzki and K. Salomaa, editors, *Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 241–250. Springer Berlin / Heidelberg, 2011.
14. S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages*, volume 1: Word, language, grammar, pages 41–110. Springer-Verlag, 1997.