Regular Expressions with Backreferences Re-examined

Martin Berglund[1,2] and Brink van der Merwe[3]

[1] Department of Information Science, Stellenbosch University, South Africa
[2] Center for AI Research, CSIR, Stellenbosch University, South Africa
[3] Department of Computer Science, Stellenbosch University, South Africa

Abstract. Most modern regular expression matching libraries (one of the rare exceptions being Google's RE2) allow backreferences, operations which bind a substring to a variable allowing it to be matched again verbatim. However, different implementations not only vary in the syntax permitted when using backreferences, but both implementations and definitions in the literature offer up a number of different variants on how backreferences match. Our aim is to compare the various flavors by considering the formal languages that each can describe, resulting in the establishment of a hierarchy of language classes. Beyond the hierarchy itself, some complexity results are given, and as part of the effort on comparing language classes new pumping lemmas are established, and old ones extended to new classes.

1 Introduction

Regular expressions as used and implemented in practice are vastly different from their traditional theoretic counterpart, both in semantics (driven by the features offered), and expectations of performance. Even when not using the more complex features the performance profile of practical regular expression matching is a fairly deep subject matter, which has seen theoretical study only fairly recently, such as in [2] and [7]. In this paper we focus on regular expressions with backreferences (rewbr for short), an advanced feature which is available in most regular expression matching libraries. This subject matter has seen some study in the literature, we will refer frequently to [1], [4], and [5], but each paper has its own definition of a rewbr and its semantics ([4] has two), and many implementations disagree with all of them (the definition given by Aho in [1] is common however), and with each other.

A backreference is placed in a regular expression to indicate that the substring matched by some specified capturing group (where capturing group is synonymous with parenthesized subexpression), should be matched again at the position (or positions) where the backreference is placed. In the Java programming language we denote by `\i` that the substring most recently matched by the ith capturing group should be matched by the backreference again, where capturing groups are numbered from 1 onwards, based on the relative position of their left parenthesis when reading the regular expression from left to right. For example, `[0-9]+\.\d*(\d+)\1+` can be used to match recurring decimal numbers, such as 0.33, 0.818181 and 0.04555, since the subexpression `(\d+)` captures some sequence of digits in the input string and the backreference `\1+` instructs the matcher to match this sequence again, one or more times. Similarly, the regular expression `(.+)\1` matches strings of the form `ww`, i.e. producing the non-context-free reduplication property.

Long and complicated regular expressions may be hard to read and maintain as adding or removing capturing groups changes the numbers of all groups following the

Martin Berglund, Brink van der Merwe: Regular Expressions with Backreferences Re-examined, pp. 30–41.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

modification. Python's `re` module was the first to offer a solution in terms named capturing groups and backreferences — `(?P<name>group)` captures the match of the subexpression `group` into `name`, whereas a backreference to the contents of this capturing group is done with `(?P=name)`. In some implementations it is then possible to *reuse* the same label for *different* capturing groups (e.g. Python and .NET both allow naming of groups, but .NET allows reusing names where Python does not), which opens possibilities obviously not available when simply numbering capturing groups from left to right. Also, regular expression matchers use different conventions in terms of how matching is defined when encountering a backreference without having captured a substring with a label corresponding to the backreference. These subtle differences in syntax and semantics allowed in rewbr influence the classes of languages described, as well as the relative succinctness of the rewbr variants. It is thus clear that a thorough comparison of rewbr variants is needed if further study is to be possible, which forms a big part of our contribution.

This paper uses as starting point the definitions and results, on rewbr, from [1], [4], [6] and [5]. In particular, the structure of the definition of matching semantics of rewbr is taken from [6], and the pumping lemma from [4] (for rewbr) provides the intuition for our own more general pumping lemmas.

The outline of the paper is as follows. After providing the necessary notation and definitions in the next section, we first give some improvements on past complexity results (demonstrating some differences between the classes), we then develop various pumping lemmas and then describe the relationships between the language classes obtained when considering the variants of rewbr as found in theory and practice.

2 Notation and Definitions

We use Σ and Φ as finite input and backreference alphabets respectively, with these (possibly empty) alphabets being disjoint. Also, $\emptyset$ and ε denote the empty set and word respectively, and $\mathbb{N}$ the set of natural numbers including 0. To improve readability, we sometimes denote $v_1 = w_1, \ldots, v_n = w_n$ as $(v_1, \ldots, v_n) = (w_1, \ldots, w_n)$. For a string w over Σ (or any other alphabet), we denote by $|w|$ the length of w, i.e. the number of occurrences of symbols from Σ in w, and more generally, if $\Sigma' \subseteq \Sigma$, then $|w|_{\Sigma'}$ is the number of occurrences of symbols from Σ' in w. For sets A and B of strings, the concatenations $A \cdot B$ is defined as usual as $\{w_1 w_2 \mid w_1 \in A, w_2 \in B\}$ and the Kleene closure of A, denoted as A^*, as $\{\varepsilon\} \cup (\cup_{i=1}^{\infty} A^i)$, where A^i is the concatenation of A with itself using the concatenation operator $(i-1)$ times.

Definition 1. *A* regular expression with backreferences (rewbr) *over input alphabet Σ and backreference alphabet Φ is defined inductively as:*

(1) $\emptyset$, or an element of $\Sigma \cup \{\varepsilon\}$, or
(2) an expression of the form $(E_1 \mid E_2)$, $(E_1 \cdot E_2)$, or, $(E_1{}^*)$, the Kleene closure, *for any rewbr E_1 and E_2, or,*
(3) for $\phi \in \Phi$, the expression $[_\phi E]_\phi$, i.e. a capturing group *labeled by ϕ, or $\uparrow_\phi$, a* backreference *to a (possibly non-existing) capturing group labeled by ϕ.*

Let $\mathrm{rewbr}_{\Sigma,\Phi}$ denote all rewbr over input alphabet Σ and backreference alphabet Φ. The subset of rewbr obtained by using only (1) and (2) above (i.e. regular expressions over Σ without backreferences), is denoted by $\mathfrak{R}_\Sigma$.

We use Σ and Φ to indicate a generic input and backreference alphabet respectively, without stating it explicitly.

As usual, parenthesis may be elided from rewbr by using the rule that Kleene closure ' * ' takes precedence over concatenation ' · ', which takes precedence over union ' | '. In addition, outermost parenthesis may be dropped and $E_1 \cdot E_2$ abbreviated as E_1E_2. The brackets which denote a capturing group may not be elided (except if no corresponding backreference appears in the rewbr). Also, for $E \in \text{rewbr}_{\Sigma,\Phi}$, we use E^+ as abbreviation for $E \cdot E^*$.

When $E \in \mathfrak{R}_\Sigma$, the language described by E, denoted as $\mathcal{L}(E)$, is defined inductively as usual, i.e. $\mathcal{L}(\emptyset) = \emptyset, \mathcal{L}(a) = \{a\}$ for $a \in \Sigma \cup \{\varepsilon\}$, $\mathcal{L}(E_1 \,|\, E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$, $\mathcal{L}(E_1 \cdot E_2) = \mathcal{L}(E_1) \cdot \mathcal{L}(E_2)$ and $\mathcal{L}(E_1{}^*) = \mathcal{L}(E_1)^*$.

Following [6] and [5], we use *ref-words* (short for *reference words*), to define the matching semantics of rewbr, instead of using the approach of Câmpeanu et al. in [4]. Câmpeanu and his co-authors used parse trees as mechanism to describe the way in which a string is matched in terms of which substring of the input string is captured by which subexpression of the rewbr. A backreference then matches a substring that is equal to the closest matched substring w' to its left in the input string, where w' was matched/captured by a subexpression labeled by the same symbol as used by the backreference. The ref-words and parse tree approaches of arriving at matching semantics are indeed equivalent. We use the ref-words approach, explained next, since it allows us to show that the various pumping lemmas for rewbr is a direct consequence of the regular language counterpart.

Let $\Sigma_\Phi = \Sigma \cup \{\phi, (_\phi,)_\phi \mid \phi \in \Phi\}$ and $w \in (\Sigma_\Phi)^*$. Then if ϕ appears in w and no subword of the form $(_\phi u)_\phi$ appears to the left of ϕ (in w), we say that the ϕ *is unbound*. We define a function $D_\varepsilon : (\Sigma_\Phi)^* \to \Sigma^*$ by using the following steps to obtain $\mathcal{D}_\varepsilon(w)$ from w. First replace all instances of ϕ by ε if ϕ is unbound. Next replace iteratively, ϕ by u, if $(_\phi u)_\phi$ is the closest subword to the left of ϕ in w starting and ending with '$(_\phi$' and '$)_\phi$' respectively, and $u \in \Sigma^*$. Finally, delete all symbols from $\{(_\phi,)_\phi \mid \phi \in \Phi\}$. The order in which the replacements are made in the second step, has no effect on the final word obtained, and thus we may assume it is made from left to right.

On occasion we are interested in the image of a specific substring in an input string under D_ε, which obviously depends on the prefix to the left of the substring of interest. We denote by $D_{\varepsilon,w}(w')$ the string obtained by removing the prefix $D_\varepsilon(w)$ from $D_\varepsilon(ww')$.

Similarly to D_ε, we let $D_\emptyset : (\Sigma_\Phi)^* \to \Sigma^*$ be the partial function with $D_\emptyset(w)$ undefined if w contains an unbound reference, and being equal to $D_\varepsilon(w)$ otherwise. The partial function $D_{\emptyset,w}(w')$ is defined in the same way as $D_{\varepsilon,w}(w')$.

We denote by $\mathbb{B}(\Sigma_\Phi)$ the subset of $(\Sigma_\Phi)^*$ of strings with well-balanced parenthesis from $\{(_\phi,)_\phi \mid \phi \in \Phi\}$. In our exposition we only use the case where the function and partial function D_ε and $D_\emptyset$ are applied on strings which are prefixes of words in $\mathbb{B}(\Sigma_\Phi)$, and in fact, D_ε and $D_\emptyset$ are mostly applied on strings from $\mathbb{B}(\Sigma_\Phi)$.

For $E \in \text{rewbr}_{\Sigma,\Phi}$, we denote by ref-regex$(E)$ the regular expression $E' \in \mathfrak{R}_{\Sigma_\Phi}$ obtained by replacing $\uparrow_\phi, [_\phi,]_\phi$ with $\phi, (_\phi,)_\phi$ respectively.

Example 2. Let $\Sigma = \{a\}$, $\Phi = \{0, 1\}$, and $E = ([_0 a \uparrow_1]_0 [_1 \uparrow_0 \uparrow_0]_1)^*$. Then $E' = \text{ref-regex}(E) = ((_0 a 1)_0 (_1 00)_1)^*$ and for $w' = (_0 a 1)_0 (_1 00)_1 (_0 a 1)_0 (_1 00)_1 \in \mathcal{L}(E')$ we

have that $D_\varepsilon(w') = a^{12}$, which is obtained by rewriting w' as follows:

$$\begin{aligned} w' &\to ({}_0a)_0({}_10^2)_1({}_0a1)_0({}_10^2)_1 \to ({}_0a)_0({}_1a^2)_1({}_0a1)_0({}_10^2)_1 \\ &\to ({}_0a)_0({}_1a^2)_1({}_0a^3)_0({}_10^2)_1 \to ({}_0a)_0({}_1a^2)_1({}_0a^3)_0({}_1a^6)_1 \\ &\to a^{12} \end{aligned}$$

The partial function $D_\emptyset$ is undefined on all of $\mathcal{L}(E')$, with the exception of ε, since all non-empty strings in $\mathcal{L}(E')$ has $({}_0a1)_0({}_100)_1$ as prefix, in which the 1 in the substring $({}_0a1)_0$ is unbound.

Remark 3. Note that $|D_\varepsilon(w)| \leq 2^{|w|}$ (and similarly for $D_\emptyset(w)$), since each time we substitute a backreference ϕ with a substring w' (where $w' \in \Sigma^*$), we at most double the length of the string. More generally, we have $|D_{\varepsilon,w_1}(w_2)| \leq \max(1, |D_\varepsilon(w_1)|)2^{|w_2|} \leq 2^{|w_1|+|w_2|}$, where the first inequality follows from the fact that in computing $D_{\varepsilon,w_1}(w_2)$ from w_2 by substitution, we may immediately use captured substrings of w_1 (if $w_1 \neq \varepsilon$) for substitution as we process w_2 from left to right.

Definition 4. *For $E \in rewbr_{\Sigma,\Phi}$, we define the language described by E based on ε-*semantics *and $\emptyset$-semantics, and denoted by $\mathcal{L}_\varepsilon(E)$ and $\mathcal{L}_\emptyset(E)$ respectively, as follows:*

- $\mathcal{L}_\varepsilon(E) = \{D_\varepsilon(w) \mid w \in \mathcal{L}(\textit{ref-regex}(E))\}$
- $\mathcal{L}_\emptyset(E) = \{D_\emptyset(w) \mid w \in \mathcal{L}(\textit{ref-regex}(E))$ *and* $D_\emptyset(w)$ *is defined* $\}$

Definition 5. *We obtain variants of rewbr by using $\mathcal{L}_\varepsilon(r)$ or $\mathcal{L}_\emptyset(r)$ or syntactically restricting the rewbr we consider in $rewbr_{\Sigma,\Phi}$ by not allowing more than one occurrence of $[_\phi$ for each ϕ (i.e. capturing labels may not be repeated) in the rewbr we consider. The four variants are then:*

	No label repetitions	***May repeat labels***
ε-semantics	*Câmpeanu-Salomaa-Yu semi-regex [4]*	*Freydenberger-Schmid [5]*
∅-semantics	*Java, Python*	*Aho [1], Boost, PCRE, .NET*

A fifth variant is the extended (non-semi) regexes of [4], which additionally require that $\uparrow_\phi$ only occur to the right of the occurrence of $]_\phi$ in the rewbr.

When we distinguish between these variants we call them (going left-to-right top-to-bottom) semi-CSY-, FS-, Java-, and Aho-style, with the addition of CSY-style to refer to the full (non-semi) regexes of [4]. We denote by $\mathbb{L}_x$ the class of languages matched by an x-style variant rewbr.

Example 6. The expression $\uparrow_1[_1\Sigma^*]_1$ can be interpreted as a semi-CSY-, FS-, Java-, or Aho-style rewbr, but not a CSY-style one (as $\uparrow_1$ occurs before $]_1$). However, the semi-CSY- and FS-style rewbr described by that expression matches Σ^*, whereas the Java- and Aho-style ones match $\emptyset$, by the difference between ε- and $\emptyset$-semantics set out in Definition 4. Meanwhile the expression $([_1a^*]_1\,|[_1(b\,|\,c)]_1)\uparrow_1$ can only describe either an FS- or Aho-style rewbr (as it repeats labels), but in this instance they match the same language, as the final $\uparrow_1$ always refers to something bound, eliminating the distinction between D_ε and $D_\emptyset$.

Remark 7. The (CSY- and Java-style) restriction of not allowing repeated labels, leads to unnatural closure properties of the respective classes of rewbr, since if $E = F(G\,|\,H)$ is of CSY- or Java-style, and F contains a capturing group, then in $E' = (FG\,|\,FG)$ a label is repeated, and E' is thus not CSY- or Java-style.

Remark 8. The additional restriction used to obtain CSY-style rewbr can be used in conjunction with the other four variants to obtain eight variants of rewbr in total, but by doing so, we end up with an additional three variants which appear to not have been considered before in literature, nor been used in practical matching software, making them of little interest to us.

Remark 9. In [5], Freydenberger and Schmid disallowed rewbr with subexpressions of the form $[_\phi \cdots \uparrow_\phi \cdots]_\phi$ (i.e. backreferences within a capturing group using the same label), since their memory automaton model, which provides a state machine equivalent formalism for the class of languages equivalent to FS-style rewbr (with this additionally stated constraint), has a memory location for each capture symbol, but it is not possible to update a memory location (of a memory automaton) and use its previous content at the same time. We, however, do not consider this restriction.

Remark 10. Notice that rewbr (independent of the choice of variant from Definition 5) are exponentially more succinct than regular expressions for some languages, for example the family

$$E_n = [_0 a]_0 [_1 \uparrow_0 \uparrow_0]_1 \cdots [_n \uparrow_{n-1} \uparrow_{n-1}]_n$$

has $\mathcal{L}(E_n) = \{a^{2^{n+1}-1}\}$, which is exponential in the length of the expression itself. By contrast, a regular expression is always at least as long as the shortest string it matches.

Next we define a generalization of the syntactic constraint that was used to define the CSY subclass of semi-CSY-style rewbr.

Definition 11. *For $E \in rewbr_{\Sigma,\Phi}$, we define the relation $\sim_E$ on Φ as $\phi \sim_E \phi'$ for $\phi, \phi' \in \Phi$, if E contains a subexpression of the form $[_\phi \cdots \uparrow_{\phi'} \cdots]_\phi$. Let $\approx_E$ be the transitive closure of $\sim_E$. Then E is* non-circular *if it is not the case that $\phi \approx_E \phi$ for any $\phi \in \Phi$.*

In a similar way as in the definition above, we define when strings in $B(\Sigma_\Phi)$ are non-circular, and note that if $w \in \mathcal{L}(\text{ref-regex}(E))$, with E non-circular, then w is also non-circular. Note that the rewbr in Example 2 is circular, while E_n in Remark 10 is non-circular.

Remark 12. The class of CSY-style rewbr has the unnatural closure (or more precisely, non-closure) property that if we start with E of CSY-style and replace in E a subexpression of the form $(F_1|F_2)$ by $(F_2|F_1)$ to obtain E', then E' might no longer be of CSY-style (but of course still semi-CSY-style). This makes it clear that non-circular rewbr (or non-circular semi-CSY) is a more natural subclass of rewbr to consider.

3 The Complexity of Backreference Matching

It is shown already in [1] that matching a rewbr to a string is NP-complete in general. In that proof a reduction from Vertex Cover is performed, with a large alphabet. As usual the alphabet can be reduced to a binary one by straightforward encoding of symbols, but we take one step further and prove that the matching problem for rewbr is NP-complete even for a unary alphabet.

Theorem 13. *Uniform membership testing a rewbr (independent of the choice of semantics from Definition 5) over alphabet Σ is NP-complete even for $|\Sigma| = 1$.*

Proof. We demonstrate this by a reduction from SATISFIABILITY (deciding satisfiability of propositional formulas on conjunctive normal form). For any instance of such a formula, $c_1 \wedge \cdots \wedge c_n$ over the variables $x_1, \ldots, x_m$, first, for each clause c_i construct the rewbr r_i as the union of backreferences for every literal in the disjunction. That is, if $c_i = x_3 \vee \overline{x_7} \vee \overline{x_9}$ (where $\overline{x}$ represents the literal negating the variable x, here viewed as a single symbol) then $r_i = \uparrow_{x_3} | \uparrow_{\overline{x_7}} | \uparrow_{\overline{x_9}}$. Then construct the rewbr:

$$R = ([_{x_1} a]_{x_1} | [_{\overline{x_1}} a]_{\overline{x_1}}) \cdots ([_{x_m} a]_{x_m} | [_{\overline{x_m}} a]_{\overline{x_m}}) r_1 \cdots r_n$$

We then argue that $a^{m+n} \in \mathcal{L}(R)$ if and only if the formula is satisfiable. This is straightforward: clearly at most $m + n$ symbols can be read (as the expression is a concatenation of $m + n$ unions). The initial sequence of unions corresponding to variables will read m symbols, in the process defining a capture *either* x_i *or* $\overline{x_i}$ for each i. The only way to read another n symbols is if every union contains at least one backreference to a literal which was chosen in the first phase. This corresponds precisely to assigning truth values to the variables, and requiring that each disjunction in the original formula contains at least one true literal.

The problem is *in* NP for all alphabet sized by a straightforward search argument over the expression. Starting at the left hand side of the expression nondeterministically search for a path through the expression in the obvious way, consuming symbols from the string as needed. If the right of the expression is reached with the entire string consumed the search accepts. This search can be restricted to polynomially many steps by rejecting whenever it would visit a position in the expression twice without either consuming a symbol or passing through a previously unvisited capturing group (i.e. defining $\uparrow_\phi$ for some ϕ where it was previously undefined) in an intervening step. The latter case is necessary for Java- and Aho-style semantics when matching e.g. $([_1 a^*]_1 | [_2 b^*]_2)^* \uparrow_1 \uparrow_2 c$ to the string c, having to repeat the first Kleene closure twice to get $\uparrow_1$ and $\uparrow_2$ initialized. This easily gives a bound of $|E|^2|w|$ (heavily overestimating), as there are $|E|$ positions, and no more than $|E|$ capturing groups which may get defined. □

Using the above result we can further demonstrate that some of the rewbr semantics we consider also give rise to a difficult emptiness problem.

Theorem 14. *For Java- and Aho-style semantics uniform membership testing and emptiness checking is NP-complete, even for $\Sigma = \emptyset$.*

Proof. For the empty alphabet emptiness checking and membership testing is equivalent, as the only string that can be in the language matched is ε. Use the same reduction that was shown in Theorem 13, but remove all as from the rewbr R. Now $\varepsilon \in \mathcal{L}_\emptyset(R)$ if and only if the formula is satisfiable.

This is easy to see, as Java- and Aho-style rewbr have the $\emptyset$-semantics defined by the $D_\emptyset$ function, which does not permit $\uparrow_{x_i}$ to match anything if it is unbound (i.e. if the capturing group labeled x_i has not been matched to something). Therefore each clause must contain some literal chosen to match in the first part of the expression (despite the captures simply being the empty string), which again simulates assigning truth values to the variables.

Membership is in NP for all alphabet sizes by the argument in Theorem 13. Emptiness is also in NP by a similar search argument, simply ignoring what symbols

are being consumed. As some expressions may contain only long strings (see e.g. Remark 10) a witness string must not be explicitly constructed, but it is sufficient for the search to track which capturing groups have been visited, capturing *some* string, not caring *which*. □

Remark 15. It should be clear that (semi-)CSY-/FS-style semantics have linear-time emptiness-checking (and therefore membership testing with $|\Sigma| = 0$), as an expression is only empty if it is a concatenation with one empty sub-expression, or is a union with both sub-expressions empty, or it equals $\emptyset$. In practical implementations $\emptyset$ is seldom even available, as it has very limited usefulness, making practical emptiness-checking constant time, since then no CSY-/FS-style expression is empty).

It is reasonably obvious, by practical use if nothing else, that the difficulty of rewbr matching is not insurmountable. If used with care, capturing in contexts where the ambiguity is low (i.e. the number of options for capturing is limited) the performance impact can be minimized. Practical regular expression libraries (all mentioned here) often have operators specifically aimed at managing such ambiguity, see for example [3]. A deeper study of the fixed parameter complexity of matching will, however, be left as future work in this paper.

4 Pumping Lemmas with Backreference Matching

The pumping lemma given in [4] is a useful tool for finding languages that cannot be matched by CSY-style rewbr. It is used in the next section to show that $\mathbb{L}_{\text{CSY}} \subsetneq \mathbb{L}_{\text{semi-CSY}}$. First we recall the definition of the pumping lemma, which we will then consider in the context of the additional semantics treated here, to then introduce a more generalized pumping lemma.

Lemma 16 (from [4]). *For every $L \in \mathbb{L}_{CSY}$ (i.e. any language matched by some CSY-style rewbr) there exists a constant k such that if $w \in L$ with $|w| > k$, then there is a decomposition $w = x_0 v x_1 v x_2 \cdots v x_n$, for some $n \geq 1$, such that:*

- $|x_0 v| < k$;
- $|v| \geq 1$; *and,*
- $x_0 v^i x_1 v^i x_2 \cdots v^i x_n \in \mathcal{L}(E)$ *for all* $i \geq 1$.

First we note that this pumping lemma does not apply to most of the other styles considered here. We satisfy ourselves with proving that it does not hold for semi-CSY-style, extending the proof to FS- and Aho-style is straightforward, but Lemma 22 will later on achieve the same result by demonstrating that languages matched by semi-CSY-style forms a subclass of FS- and Aho-style.

Lemma 17. *The pumping lemma of [4] does not hold for semi-CSY-style rewbr.*

Proof. This follows from there being exponentially growing languages matched by semi-CSY-style rewbr. Let $E = ([_\alpha\uparrow_\beta\uparrow_\beta]_\alpha[_\beta a\alpha]_\beta)^*$. Then $L = \mathcal{L}_\varepsilon(E) = \{a^{2^n-1} \mid n \geq 1\}$ and $L \in \mathbb{L}_{\text{semi-CSY}}$. The result now follows by observing that the pumping lemma recalled in Lemma 16 does not hold for L, as it implies there would exist some k, n and v such that $a^{k+i \cdot n|v|} \in L$ for $i \geq 1$, which precludes strict exponential growth. □

However, this pumping lemma does hold for Java-style rewbr.

Lemma 18. *The pumping lemma of [4] also holds for Java-style rewbr.*

Proof. The intuition for why Java-style rewbr differs from semi-CSY-style in this regard is that, while there may be circular capturing groups in Java, the first capture in the cycle must be possible to perform without using any of the other backreferences in the cycle (as they will be unbound). Since the capturing labels cannot repeat, the option of not using any backreference in the capturing sub-expression will then remain on every subsequent repetition of the cycle, making it possible to "restart" the cycle at will. A formal argument follows.

Let E be a Java-style rewbr, set $k = 2^{|E|}$, to match a string w with $|w| > k$ some Kleene closure must be repeated at least once (matching a backreference may at most double the length of the string matched, see e.g. Remark 3, and, obviously, E contains at most linearly many backreferences). Fix one particular match for w, and take the Kleene closure which first repeats a full match of its enclosed subexpression, $x_0 v x_1 v x_2 \cdots v x_n$, where the first v is the substring matched by the the first repetition of the F subgroup, and each following v is produced by a backreference to that initial matching of the subgroup (obviously n may be one if the capture is never referred to). Then we argue that in Java-style semantics $x_0 v^i x_1 v^i x_2 \cdots v^i x_n \in \mathcal{L}(E)$ for all $i \geq 1$.

This is the case as, by assumption, the match of v was the first entry into F, and as unbound backreferences do not match in Java-style, and capture group labels may not repeat, this means the match of v used a path through F on which no backreference is used which is subsequently assigned by a capture inside F (as such a backreference would have had to be undefined). This means that if F is repeated, it will be able to match v again, any number of times, without changing any capturing group contents (performing the same captures as it did the first time), using the same path through F in each instance. The remaining vs, down the line, are produced by backreferences and need no special argument. □

In the next three lemmas we develop a more general pumping lemma for $\mathbb{L}_{\text{CSY}}$.

Lemma 19. *For $L \in \mathbb{L}_{CSY}$ there exists a constant k_L such that if $w_0 w \in L$ with $|w| \geq k_L \max(1, |w_0|)$, then we have strings u, x, y, z such that $uxyz, y \in \mathbb{B}(\Sigma_\Phi)$, with:*

- $(D_\varepsilon(u), D_{\varepsilon,u}(xyz)) = (w_0, w)$ *and thus* $D_\varepsilon(uxyz) = w_0 w$;
- $|D_{\varepsilon,u}(xy)| \leq k_L \max(1, |w_0|)$;
- $D_{\varepsilon,uxy^i}(y) = D_{\varepsilon,ux}(y) \neq \varepsilon$ *for all* $i \geq 0$; *and*
- $D_\varepsilon(uxy^*z) \subseteq L$.

Proof. Assume $L = \mathcal{L}_\varepsilon(E)$, $E' = \text{ref-regex}(E)$, $p > |E'|$ (i.e. p is a pumping constant for the regular language $\mathcal{L}(E')$) and $k_L = 2^p$. The result now follows from the relationship between regular languages and $\mathbb{L}_{\text{CSY}}$ via the function D_ε, Remark 3 and the pumping lemma applied to the regular language $\mathcal{L}(E')$. Next the details. We have u, u' with $(D_\varepsilon(u), D_{\varepsilon,u}(u')) = (w_0, w)$ and $uu' = \mathcal{L}(E')$. From Remark 3, $|u'| \geq p$. The pumping lemma for $\mathcal{L}(E')$ implies we have x, y, z with $u' = xyz$, $uxy^*z \subseteq \mathcal{L}(E') \subseteq \mathbb{B}(\Sigma_\Phi)$ and thus $D_\varepsilon(uxy^*z) \subseteq L$. We may assume y is matched by F, with F^* a subexpression of E' and F not containing Kleene stars. To get $|D_{\emptyset,ux}(y)| \geq 1$, we consider all possible non-empty substrings y of u' matched by an F with F^* a subexpression of E', and if for all of them $|D_{\emptyset,ux}(y)| = 0$, we would get a contradiction to $|w| \geq k_L \max(1, |w_0|)$. By picking the first possible y (to the left) with $|D_{\emptyset,ux}(y)| \geq 1$, we also ensure $|D_{\varepsilon,u}(xy)| \leq k_L \max(1, |w_0|)$, by using Remark 3.

Finally, to obtain $D_{\varepsilon,uxy^i}(y) = D_{\varepsilon,ux}(y)$ for all $i \geq 0$, we need the CSY assumption that backreferences do not appear before corresponding capturing subexpressions

in E, which implies that if $y = y_1\phi y_2$, then y_1 contains a substring of the form $({}_\phi y')_\phi$. This is enough to ensure $D_{\varepsilon,uxy^i}(y) = D_{\varepsilon,ux}(y)$. □

Lemma 20. *Assume x, y, z are strings with $xyz, y \in \mathbb{B}(\Sigma_\Phi)$ and xyz non-circular with $(D_\varepsilon(x), D_{\varepsilon,x}(y)) = (x_0, v)$. Also assume if $y = y_1\phi y_2$, then y_1 contains a substring of the form $({}_\phi y')_\phi$. Then for $i \geq 1$, $D_\varepsilon(xy^iz) = x_0v^ix_1v^ix_2 \cdots v^ix_n$, for some strings $x_1, \ldots, x_n$ where $n \geq 1$.*

Proof. If $xyz = w_0({}_\phi w_1 y w_2)_\phi w_3$, where w_3 has $(n-1)$ occurrences of the symbol ϕ before any substring of the form $({}_\phi w)_\phi$ (which includes the case of w_3 not having a substring of the form $({}_\phi w)_\phi$), then $D_\varepsilon(xy^iz)$ is as specified. Otherwise, if y is not properly contained in a substring of the form $({}_\phi w)_\phi$, we have $n = 1$. □

To see that the non-circular requirement is necessary in Lemma 20, take $x = z = \varepsilon$ and $y = ({}_0a1)_0({}_100)_1$. Then $D_\varepsilon(y) = a^3, D_\varepsilon(y^2) = a^{12}$, $D_\varepsilon(y^3) = a^{33}$, and in general, $|D_\varepsilon(y^i)| \geq 3^i$. Also, if $(x, y, z) = (({}_0a)_0, 0({}_0b)_0, \varepsilon)$, then $D_\varepsilon(xyz) = a^2b$, while $D_\varepsilon(xy^{i+1}z)) = a^2bb^{2i}$, and thus the reason for assuming if $y = y_1\phi y_2$, then y_1 contains a substring of the form $({}_\phi y')_\phi$.

Lemma 21. *For $L \in \mathbb{L}_{CSY}$ there is a constant k_L such that if $w_0w \in \mathbb{L}$ with $|w| \geq k_L \max(1, |w_0|)$, then there are strings $x_0, \ldots, x_n, v$, for some $n \geq 1$, with $|v| \geq 1$, so that we have:*

- *$w = x_0vx_1vx_2 \cdots vx_n$; and*
- *$w_0x_0v^ix_1v^ix_2 \cdots v^ix_n \in L$ for all $i \geq 1$.*

Proof. From Lemma 19 we have u, x, y, z with $(D_\varepsilon(u), D_{\varepsilon,u}(xyz)) = (w_0, w)$ (and $D_\varepsilon(uxyz) = ww_0$), $D_{\varepsilon,uxy^i}(y) = v \neq \varepsilon$ for $i \geq 0$, with $D_\varepsilon(uxy^*z) \subseteq L$. $L \in \mathbb{L}_{\text{CSY}}$ implies we can use Lemma 20 and conclude that there is some $n \geq 1$ so that $D_\varepsilon(uxy^iz) = w_0x_0v^ix_1v^ix_2 \cdots v^ix_n \in L$, for $i \geq 1$. □

Beyond its general usefulness, Lemma 21 will be used to distinguish between the language classes matched by CSY- and Java-style rewbr in Lemma 25.

5 Language Hierarchies

In the previous section several containment relationships between the language classes which can be matched by the different styles of rewbr were established, in this section we refine this further. Let us begin by combining and summarizing a few straightforward relations with what was already established in previous sections.

Lemma 22. *The following inclusions hold: $\mathbb{L}_{CSY} \subsetneq \mathbb{L}_{semi\text{-}CSY} \subseteq \mathbb{L}_{FS} \subseteq \mathbb{L}_{Aho}$, and in addition, $\mathbb{L}_{semi\text{-}CSY} \not\subseteq \mathbb{L}_{Java} \subseteq \mathbb{L}_{Aho}$.*

Proof. $\mathbb{L}_{\text{CSY}} \subseteq \mathbb{L}_{\text{semi-CSY}} \subseteq \mathbb{L}_{\text{FS}}$ and $\mathbb{L}_{\text{Java}} \subseteq \mathbb{L}_{\text{Aho}}$ follow directly by Definition 5, by explicit restrictions placed on the styles. $\mathbb{L}_{\text{semi-CSY}} \not\subseteq \mathbb{L}_{\text{CSY}}$ and $\mathbb{L}_{\text{semi-CSY}} \not\subseteq \mathbb{L}_{\text{Java}}$ is shown in Lemma 17. Finally, $\mathbb{L}_{\text{FS}} \subseteq \mathbb{L}_{\text{Aho}}$ since Aho-style can simulate FS-style, i.e. $\mathbb{L}_{\text{FS}} \subseteq \mathbb{L}_{\text{Aho}}$, since if $E \in \text{rewbr}_{\Sigma,\Phi}$ and $E' = [_{\phi_1}]_{\phi_1} \cdots [_{\phi_n}]_{\phi_n} E$, where $\uparrow_{\phi_1}, \ldots, \uparrow_{\phi_n}$ are all of the distinct backreference symbols in E, then $\mathcal{L}_\varepsilon(E) = \mathcal{L}_\emptyset(E')$. □

As a first additional piece of the puzzle we demonstrate that the two most powerful formalisms in the hierarchy are actually equivalent.

Lemma 23. $\mathbb{L}_{FS} = \mathbb{L}_{Aho}$.

Proof. Lemma 22 already demonstrates that $\mathbb{L}_{\text{FS}} \subseteq \mathbb{L}_{\text{Aho}}$, so all that is still needed is to establish that $\mathbb{L}_{\text{Aho}} \subseteq \mathbb{L}_{\text{FS}}$. Let $\mathcal{A}$ be Aho-style rewbr E with the property that if $w \in \mathcal{L}(\text{ref-regex}(E))$, then w has no unbound reference. Thus $\mathcal{L}_\emptyset(E) = \mathcal{L}_\varepsilon(E)$ for $E \in \mathcal{A}$. Let F be a rewbr of Aho-style. We show that there exists $F' \in \mathcal{A}$ with $\mathcal{L}_\emptyset(F) = \mathcal{L}_\emptyset(F') = \mathcal{L}_\varepsilon(F')$ and thus $\mathbb{L}_{\text{Aho}} \subseteq \mathbb{L}_{\text{FS}}$. Let $[_{\phi_1}F_1]_{\phi_1}, \ldots, [_{\phi_k}F_k]_{\phi_k}$ be all capturing subexpressions in F. We replace subexpressions in F as follows:

- subexpressions of the form $\bar{F}^*$ are replaced by $(\varepsilon \mid \bar{F}^+)$; and
- $(\bar{F} \mid \bar{G})^*$ is replaced by $\bar{F}^+\bar{G}(\bar{F} \mid \bar{G})^* \mid \bar{G}^+\bar{F}(\bar{F} \mid \bar{G})^*)$.

After these replacements we use the fact that concatenation distribute over union to obtain $F' = (H_1 \mid \ldots \mid H_l)$, with $\mathcal{L}_\emptyset(F) = \mathcal{L}_\emptyset(F')$. Each H_i is such that if $\uparrow_{\phi_k}$ is a subexpression of H_i, then H_i is a concatenation of subexpressions, one of which is $H' = [_{\phi_k}F_k]_{\phi_k}$, and this subexpression H' appears in the concatenation of subexpressions forming H_i, before the subexpression containing $\uparrow_\phi$. Thus during a match with H_i, a subexpression of the form $[_{\phi_k}F_k]_{\phi_k}$ must be used to match a substring of the input string, before encountering $\uparrow_{\phi_k}$. This property of the H_i's ensures that $F' \in \mathcal{A}$. □

Example 24. Here we illustrate part of the construction used in the proof of Lemma 23 to turn an Aho-style rewbr into a language equivalent FS-style rewbr. Let Π_n be the set of $n!$ permutations on $\{1, \ldots, n\}$, and $P_n = \{a_{\pi(1)} \cdots a_{\pi(n)} \mid \pi \in \Pi_n\}$. Then $\mathcal{L}_\emptyset(E_n) = P_n$, where E_n is the Aho-style (in fact Java-style) rewbr given by:

$$(a_1({}_1\varepsilon)_1 \mid \ldots \mid a_n({}_n\varepsilon)_n)^n \uparrow_{\phi_1} \ldots \uparrow_{\phi_n}$$

Since E_n contains no Kleene star operators, when we use the procedure described in the proof of the previous theorem, we simply have to distribute concatenation over union to obtain an FS-style (more precisely, semi-CSY-style) rewbr F_n, with $\mathcal{L}_\emptyset(E_n) = \mathcal{L}_\varepsilon(F_n)$. When we do this, we obtain that F_n is the union of $n!$ subexpressions of the following form, for all $\pi \in \Pi_n$:

$$a_{\pi(1)}({}_{\pi(1)}\varepsilon)_{\pi(1)} \cdots a_{\pi(n)}({}_{\pi(1)}\varepsilon)_{\pi(n)} \uparrow_{\phi_1} \cdots \uparrow_{\phi_n}$$

Note that when distributing concatenation over union, we get many more subexpressions which are all of form $b_1 \cdots b_n \uparrow_{\phi_1} \cdots \uparrow_{\phi_n}$, with $b_i \in \{a_1, \ldots, a_n\}$, for $1 \leq i \leq n$. But when some of the b_i's are equal (i.e when we have backreferences to non-existing capturing groups), the languages represented by these subexpressions are empty in Aho-style, and they are thus not used in F_n. The semi-CSY-style rewbr F_n is of course more complicated than necessary to describe P_n, but it remains open if a more succinct rewbr of semi-CSY-style or even FS-style exists for the language P_n, than simply taking the union of all $n!$ subexpressions of the form $a_{\pi(1)} \cdots a_{\pi(n)}$.

Further, while Java does fulfill the original pumping lemma recalled in Lemma 16, it does in fact not fulfill Lemma 21, the generalized pumping lemma for $\mathbb{L}_{\text{CSY}}$.

Lemma 25. $\mathbb{L}_{Java} \not\subseteq \mathbb{L}_{CSY}$.

Proof. Take the Java-style rewbr $E = ([_0a \mid \uparrow_0 a]_0 b)^*$, for which we have $\mathcal{L}(E) = (\{(ab)^i \mid i \geq 0\} \cup \{a^{2^0}b \cdots a^{2^i}b \mid i \geq 0\})^*$. We argue that Lemma 21 does not hold for $\mathcal{L}(E)$, by assuming the contrary and taking $w_0 = ab$ and $w = a^{2^1}b \cdots a^{2^j}b$, with $|w| \geq 2k_{\mathcal{L}(E)}$. Let u be the non-empty pumping substring. Clearly any u consisting

of only as does not work, but choosing a string containing a b will under pumping also give rise to a substring of the form $\cdots a^l b a^l b \cdots$, which is not in $\mathcal{L}(E)$ for any l except for $l = 1$, however, that corresponds to $u = ab$, and the only place where that substring could be pumped in this particular string would be in the initial prefix, but as that prefix is taken by w_0 that is not available as a choice. We thus conclude that $\mathbb{L}_{\text{Java}} \not\subseteq \mathbb{L}_{\text{CSY}}$. □

We are with these results in hand ready to summarize the containment results for the classes of languages matched by the various variants of rewbr.

Theorem 26. *The following inclusions hold.*

$$\mathbb{L}_{CSY} \subsetneq \mathbb{L}_{semi\text{-}CSY} \subsetneq \mathbb{L}_{FS} = \mathbb{L}_{Aho}$$
$$\mathbb{L}_{CSY} \not\supseteq \mathbb{L}_{Java} \subsetneq \mathbb{L}_{FS}$$

Proof. Combine Lemmas 22 (in turn using Lemma 17), 23, and 25.

6 Conclusions and Future Work

These initial definitions, pumping lemmas, and inclusion proofs create a solid foundation, there are still numerous avenues for further investigation available:

- The inclusions of Theorem 26 paint a fairly clear picture, but it does remain to show whether the non-inclusions are one side of the classes being incomparable, or, seemingly more likely, whether they can be expanded into containments. That is, it seems a reasonable conjecture that a completed result should read

  $$\mathbb{L}_{\text{CSY}} \subsetneq \mathbb{L}_{\text{Java}} \subsetneq \mathbb{L}_{\text{semi-CSY}} \subsetneq \mathbb{L}_{\text{FS}} = \mathbb{L}_{\text{Aho}},$$

  but the actual inclusions of $\mathbb{L}_{\text{CSY}}$ in $\mathbb{L}_{\text{Java}}$ and $\mathbb{L}_{\text{Java}}$ in $\mathbb{L}_{\text{semi-CSY}}$ remain to be demonstrated.
- Pumping lemmas which generalize to semi-CSY-, FS- and Aho-style rewbr should also be found, it may be that Lemma 19 can be adapted to these cases with some minor restatements and additional argument.
- While differences between the language classes matched seems the most important point from a theoretical perspective, it may for practical purposes be almost more important to determine the relative succinctness of the rewbr variants. The exponential growth exhibited by some of the classes, demonstrating the differences, is likely not languages of very great interest for practical matching. However, how compactly some of the languages within the intersection of the language classes can be described could inform choices for future implementations (e.g. if Aho-style is not more succinct on interesting cases it may be inadvisable to accept the additional power offered by the variant).
- Finally, the most important practical questions is no doubt matching time complexity. While we give a refinement on the hardness of matching with rewbr in Section 3 there is much more that can be done attacking this problem. Applying parameterized complexity theory to study which aspects of the problem cause the seeming high complexity seems a promising avenue, as practical use suggests that suitably limited use of backreferences make for matching performance which is in fact entirely tractable.

More broadly the area of practical regular expressions remains teeming with poorly understood extensions and common use cases which require study to form a solid theoretical foundation for practical string matching.

References

1. A. V. Aho: in Handbook of Theoretical Computer Science (Vol. A), J. van Leeuwen, ed., MIT Press, Cambridge, MA, USA, 1990, ch. Algorithms for Finding Patterns in Strings, pp. 255–300.
2. M. Berglund, F. Drewes, and B. van der Merwe: *Analyzing catastrophic backtracking behavior in practical regular expression matching*, in Automata and Formal Languages, Z. Ésik and Z. Fülöp, eds., vol. 151 of Electronic Proceedings in Theoretical Computer Science, 2014, pp. 109–123.
3. M. Berglund, B. van der Merwe, B. Watson, and N. Weideman: *On the semantics of atomic subgroups in practical regular expressions*, in Implementation and Application of Automata, A. Carayol and C. Nicaud, eds., vol. 10329 of Lecture Notes in Computer Science, Springer, 2017, pp. 14–26.
4. C. Câmpeanu, K. Salomaa, and S. Yu: *A formal study of practical regular expressions.* International Journal of Foundations of Computer Science, 14(6) 2003, pp. 1007–1018.
5. D. D. Freydenberger and M. L. Schmid: *Deterministic regular expressions with back-references*, in 34th Symposium on Theoretical Aspects of Computer Science (STACS 2017), H. Vollmer and B. Vallée, eds., vol. 66 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2017, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 33:1–33:14.
6. M. L. Schmid: *Characterising REGEX languages by regular languages equipped with factor-referencing*, in Developments in Language Theory - 18th International Conference, DLT 2014, Ekaterinburg, Russia, August 26-29, 2014. Proceedings, A. M. Shur and M. V. Volkov, eds., vol. 8633 of Lecture Notes in Computer Science, Springer, 2014, pp. 142–153.
7. N. Weideman, B. van der Merwe, M. Berglund, and B. Watson: *Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA*, in Implementation and Application of Automata, Y. Han and K. Salomaa, eds., vol. 9705 of Lecture Notes in Computer Science, Springer, 2016, pp. 322–334.