MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 31, NO. 4, 2024
journal homepage: www.mais-journal.ru

Disambiguation of Regular Expressions With Backreferences via Term Rewriting

D. N. Ismagilova[1], A. N. Nepeivoda[2]

DOI: 10.18255/1818-1015-2024-4-426-445

[1]Bauman Moscow State Technical University, Moscow, Russia
[2]Program Systems Institute of RAS, Veskovo, Yaroslavl region, Russia

MSC2020: 68Q45
Research article
Full text in English

Received November 6, 2024
Revised November 15, 2024
Accepted November 30, 2024

In this paper we focus on regular expressions with acyclic backreferences and treat them as a semiring satisfying certain theorems of Kleene algebra. Using these theorems as term rewriting rules, we introduce an algorithm for memory disambiguation of regular expressions. Furthermore, we demonstrate that the class of regexes with acyclic backreferences is closed under language reversal, in contrast to the generic backref-regexes, and provide the reversal algorithm, based on the disambiguation procedure. The results of our experiments revealed that, in certain cases, the matching time was significantly reduced when using the reversed expressions compared to the initial ones.

Keywords: extended regular expression; backreferences; Kleene algebra; capture group; reversal, disambiguation

INFORMATION ABOUT THE AUTHORS

Ismagilova, Daria N.	ORCID iD: 0009-0000-9915-3039. E-mail: armi.din1902@gmail.com Undergraduate
Nepeivoda, Antonina N. (corresponding author)	ORCID iD: 0000-0003-3949-2164. E-mail: a_nevod@mail.ru Researcher

Funding: Russian Academy of Sciences (Project no. 122012700089-0).

For citation: D. N. Ismagilova and A. N. Nepeivoda, “Disambiguation of Regular Expressions with Backreferences via Term Rewriting”, *Modeling and Analysis of Information Systems*, vol. 31, no. 4, pp. 426–445, 2024. DOI: 10.18255/1818-1015-2024-4-426-445.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 31, № 4, 2024
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Устранение неоднозначностей в расширенных регулярных выражениях с обратными ссылками посредством применения правил переписывания

Д. Н. Исмагилова[1], А. Н. Непейвода[2]

DOI: 10.18255/1818-1015-2024-4-426-445

[1]Московский государственный технический университет им. Н. Э. Баумана, Москва, Россия
[2]Институт Программных Систем им. А.К. Айламазяна РАН, Веськово, Ярославская обл., Россия

УДК 004.415.52
Научная статья
Полный текст на английском языке

Получена 6 ноября 2024 г.
После доработки 15 ноября 2024 г.
Принята к публикации 30 ноября 2024 г.

В работе рассматривается класс расширенных регулярных выражений с обратными ссылками, которые представляются как элементы полукольца, частично удовлетворяющего теоремам алгебры Клини. Используя эти теоремы в качестве правил переписывания, возможно построить алгоритм устранения неоднозначности в ячейках памяти выражений. В дальнейшем этот алгоритм может быть применён для построения обращений расширенных регулярных выражений в заданных ограничениях. Предложенные алгоритмы были апробированы на тестовой выборке регулярных выражений, построенных на базе выражений из RegexLib и StackOverflow. Результаты экспериментов показали, что в ряде случае время сопоставления с преобразованным регулярным выражением было значительно меньше, чем с исходным.

Ключевые слова: расширенные регулярные выражения; обратные ссылки; группы захвата; реверсирование; неоднозначность; алгебра Клини

ИНФОРМАЦИЯ ОБ АВТОРАХ

Исмагилова, Дарья Наильевна	ORCID iD: 0009-0000-9915-3039. E-mail: armi.din1902@gmail.com Бакалавр
Непейвода, Антонина Николаевна (автор для корреспонденции)	ORCID iD: 0000-0003-3949-2164. E-mail: a_nevod@mail.ru Научный сотрудник

Финансирование: Российская академия наук (НИР 122012700089-0).

Для цитирования: D. N. Ismagilova and A. N. Nepeivoda, “Disambiguation of Regular Expressions with Backreferences via Term Rewriting”, *Modeling and Analysis of Information Systems*, vol. 31, no. 4, pp. 426–445, 2024. DOI: 10.18255/1818-1015-2024-4-426-445.

Introduction

Regular expressions in the classical textbooks on formal language theory are defined as a set of expressions closed under alternation, iteration, and concatenation. These expressions define regular languages, i.e. they are equivalent to the finite state automata. In modern formal language theory, the notion "regex" is used to define *extended* Perl-like regular expressions, making use of capture groups and backreferences. For the sake of clarity, the classical regular expressions are usually[1] called "academic" [1].

There exist well-known algorithms efficiently matching strings against academic regular expressions. These algorithms are implemented in regex libraries that support the academic regex style, such as Rust regex module or RE2 library[2]. However, implementing the classical approach with a transition to deterministic finite automata (DFA) can sometimes be challenging, especially when using certain syntactic sugar like unrestricted lookaheads. The lookaheads require the intersection construction, which leads to the critical growth in the DFA size in certain cases [2]. In order to address this issue, the RE2 library adopts a flexible non-determinisic-automata(NFA)–DFA approach, relying on the regex structure. For instance, if a regular expression contains no ambiguity, then it is simply transformed to Glushkov NFA [3] (the transformation produces a DFA in this case). If a regular expression is reverse-unambiguous, then its reversal is used in the matching algorithm. For example, the expression $(.^*)\,\mathtt{a}\,.\{\,\mathtt{n}\,\}$, which generates exponentially large DFAs, is treated as a reversal of deterministic regular expression $.\{\,\mathtt{n}\,\}\,\mathtt{a}\,(.^*)$.

Rewriting the regexes without transitioning to finite automata is also used in other ways to boost the matching efficiency. Conversion to the strong star-normal form, for instance, allows a matching algorithm to avoid nested-stars ambiguity as in the regex $(.^*)^*$. Since equivalence of classical regular expressions can be checked by complete axioms schemas, the corresponding schemas can be used as rewriting rules for regex optimisation.

However, in most regex libraries, including the widely-used PCRE2[3], languages defined inside the supported regex syntax are context-sensitive, which make them more expressive but also harder to analyse compared to the academic case. Since matching against such an extended regex is known to be NP-complete [4], popular algorithms for treating the PCRE2 regexes use backtracking-based matching techniques, with limited (e.g. in Java 11+) or no optimizations of the complex regex structure[4]. This results in large matching time against even relatively small strings if a regex is ambiguous. Estimating regex ambiguity is another open problem due to the context-sensitivity: in order to define determinism, one must consider values stored in memory cells corresponding to the back-references. Moreover, anonymous capture groups make it difficult to define the extended regex algebra, as the reference containment may change depending on the parentheses structure. In particular, this property ruins associativity of alternation. For example, PCRE2 expression[5] $(\mathtt{(a\mid b)\mid c})\backslash 2$ recognises language $\{\mathtt{aa}, \mathtt{bb}\}$, while expression $(\mathtt{a\mid(b\mid c)})\backslash 2$ recognises language $\{\mathtt{bb}, \mathtt{cc}\}$.

In the last decade, new matching algorithms with the backreference support were constructed, such as those based on memory finite automata (MFA) [5]. These algorithms inherit some properties of Glushkov automaton [3]. However, the MFA formalism admits non-balanced capture groups, which makes the use of rewriting rules problematic.

The goal of this work is to partly implement the techniques used in RE2 library for the extended syntax. We defined a class of extended regular expressions **ACREG** satisfying certain laws of Kleene algebra,

[1] Sometimes terms "standard" and "basic" are used to distinguish the "pure" regexes from the extended ones as well. However, a "standard regex" can be confused with "standard POSIX" or "standard PCRE" regex, and "basic regex" is overloaded also to denote the basic syntax of regular expressions with no option and positive iteration syntactic sugar.

[2] https://github.com/google/re2

[3] https://www.pcre.org/current/doc/html/index.html

[4] https://www.pcre.org/current/doc/html/index.html, https://github.com/python/cpython

[5] Since PCRE2 does not admit usage of uninitialised references, the alternation arguments not belonging to the second capture group are not recognised by the expressions.

and used these laws in order to disambiguate memory usage in the regexes. We studied a dataset of regexes taken from StackOverflow[6] and discovered that all of these regexes can be treated as elements of the **ACREG** class. Based on the memory-disambiguated structure, we provided the algorithm reversing **ACREG** regexes. We also proved that class of the generic extended regexes defined in paper [5] is not closed under reversal. The suggested algorithms were implemented in a model regex engine in order to test efficiency of matching strings against disambiguated regexes and their reversed versions. The experiments showed that the memory disambiguation makes it possible to use rather efficient matching heuristics.

The paper is structured as follows. Section 1 contains basic definitions of backref-regexes and rewriting rules based on Kleene algebra laws, memory finite automaton construction, and notions of deterministic and backwards-deterministic academic regular expressions. Section 2 describes the regex formalism used, its main properties, and theoretical concepts underlying the normalisation and reversal algorithms. Section 3 presents the experiments with implementation of the algorithms, Section 4 discusses the related works considering regex simplification and backreferences formalisms. The proofs of the main statements of the paper and the algorithms pseudocode are given in the Appendix.

1. Preliminaries

Henceforth we denote a letter alphabet by Σ, and the empty word by ε. If ρ is a regex, then $\mathscr{L}(\rho)$ is its language, i.e. the set of the words recognized by ρ.

The regexes, as well as the letters from Σ, are written in `typewriter` font.

1.1. Kleene algebra

Definition 1. A semiring $\langle \mathcal{S}, +, \cdot, 0, 1\rangle$ over $\mathcal{S}$ is a commutative monoid wrt $+$ with the unit element 0 and monoid[7] wrt $\cdot$ with the unit element 1 satisfying the additional axioms:

- $\forall x, y, z \in \mathcal{S}\big((x + y) \cdot z = x \cdot z + y \cdot z\big)$ (right distributivity)
- $\forall x, y, z \in \mathcal{S}\big(x \cdot (y + z) = x \cdot y + x \cdot z\big)$ (left distributivity)
- 0 is zero wrt $\cdot$.

Definition 2. A Kleene algebra is a semiring $\langle \mathcal{A}, +, \cdot, \emptyset, 1\rangle$, idempotent wrt $+$, with the additional unary operation *, satisfying the following axioms [7]:

- $\forall x \in \mathcal{A}\big(1 + x \cdot x^* = x^* \;\&\; 1 + x^* \cdot x = x^*\big)$ (unfolding rule)
- $\forall x, y, z \in \mathcal{A}\big((x \cdot y + z + y = y \Rightarrow x^* \cdot z + y = y) \;\&\; (y \cdot x + z + y = y \Rightarrow z \cdot x^* + y = y)\big)$ (Kozen's axioms)

Academic regular expressions over an alphabet Σ form a Kleene algebra, where union (alternation) is treated as the addition, and concatenation — as the multiplication. The following theorems are satisfied in any Kleene algebra.

$\forall x, y \in \mathcal{A}\big(x \cdot (y \cdot x)^* = (x \cdot y)^* \cdot x\big)$ *(sliding rule)*
$\forall x, y \in \mathcal{A}\big((x + y)^* = x^* \cdot (y \cdot x^*)^* = (x^* \cdot y)^* \cdot x^*\big)$ *(denesting rule)*
$\forall x \in \mathcal{A}\big(x^* = (x^n)^*(\varepsilon \mid x \mid x^2 \mid \ldots x^{n-1})\big)$ *(fusion rule)*
$\forall x, y, z \in \mathcal{A}\big(x \cdot y = y \cdot z \Rightarrow x^* \cdot y = y \cdot z^*\big)$ *(bisimulation rule)*

The denesting rule is called so because it is used to reduce the star height of an expression, through transforming iterations to unions; we use this rule in backward order, so we call it the nesting transformation.

1.2. Deterministic Regular Expressions

Definition 3. A linearization ρ_{lin} of a regex ρ is obtained by indexing all the letters from Σ in ρ by their positions.

[6] https://github.com/SBULeeLab/LinguaFranca-FSE19
[7] Following the paper [6], we assume existence of the unit wrt $\cdot$.

Table 1. FIRST and LAST sets

Regular expression	FIRST	LAST
a	{a}	{a}
$\tau_1 \mid \tau_2$	$\mathrm{FIRST}(\tau_1) \cup \mathrm{FIRST}(\tau_2)$	$\mathrm{LAST}(\tau_1) \cup \mathrm{LAST}(\tau_2)$
$\tau_1\tau_2$	$\mathrm{FIRST}(\tau_1) \cup \mathrm{FIRST}(\tau_2)$, if $\varepsilon \in \mathscr{L}(\tau_1)$; $\mathrm{FIRST}(\tau_1)$ otherwise	$\mathrm{LAST}(\tau_1) \cup \mathrm{LAST}(\tau_2)$, if $\varepsilon \in \mathscr{L}(\tau_2)$; $\mathrm{LAST}(\tau_2)$ otherwise
τ^*	$\mathrm{FIRST}(\tau)$	$\mathrm{LAST}(\tau)$

Table 2. FOLLOW set

Regular expression	FOLLOW
ab	$\{(\mathtt{a}, \mathtt{b})\}$
$\tau_1 \mid \tau_2$	$\mathrm{FOLLOW}(\tau_1) \cup \mathrm{FOLLOW}(\tau_2)$
$\tau_1\tau_2$	$\mathrm{FOLLOW}(\tau_1) \cup \mathrm{FOLLOW}(\tau_2)$ $\{(x_1, x_2) \mid x_1 \in \mathrm{LAST}(\tau_1)\ \&\ x_2 \in \mathrm{FIRST}(\tau_2)\}$
τ^*	$\mathrm{FOLLOW}(\tau) \cup \{(x_1, x_2) \mid x_1 \in \mathrm{LAST}(\tau)\ \&\ x_2 \in \mathrm{FIRST}(\tau)\}$

For example, the linearized version of expression $(\mathtt{a} \mid \mathtt{b})^*\mathtt{a}$ is the expression $(\mathtt{a}_1 \mid \mathtt{b}_2)^*\mathtt{a}_3$.

Definition 4. Given an academic regex ρ, it is called deterministic, or one-unambiguous [8], if for every two words ω_1, ω_2 in $\mathscr{L}(\rho_{\mathrm{lin}})$, s.t. $\omega_1 = u\gamma_i w$, $\omega_2 = u\gamma_j w'$ and γ_i and γ_j are linearizations of a same letter $\gamma \in \Sigma$, $i = j$ holds.

We say that regex ρ is backwards-deterministic, if its reversal (denoted with ρ^R) is deterministic.

An alternative definition of the deterministic regular expression can be formulated in terms of a Glushkov automaton [3] if Glushkov(ρ) is deterministic, then ρ is deterministic. A Glushkov non-deterministic finite automaton (NFA) for an academic regex ρ can be defined using FIRST, LAST, and FOLLOW sets of ρ_{lin} (see Tables 1, 2). The letters of ρ_{lin} correspond to non-initial states of Glushkov(ρ); the LAST-set points out the final states, the FIRST-set defines transitions from the initial state, and FOLLOW-set defines all the other transitions.

An example of the Glushkov NFA based on the regex $(\mathtt{a} \mid \mathtt{b})^*\mathtt{a}$ is given in Figure 1. This NFA is non-deterministic — there are ambiguous transitions from the state $\mathtt{a}_1$ to states $\mathtt{a}_1$ and $\mathtt{a}_3$. The reversed expression $\mathtt{a}(\mathtt{a} \mid \mathtt{b})^*$ produces a deterministic Glushkov NFA, thus, $(\mathtt{a} \mid \mathtt{b})^*\mathtt{a}$ is backwards-deterministic.

The RE2 library utilises both properties of regex determinism and backwards-determinism, in order to choose the most efficient NFA representation[8].

1.3. Regular Expressions with Backreferences

In general, the extended regex syntax is borrowed from the paper [5]. Following that paper, we also call such expression *ref-words*.

[8] https://github.com/google/re2

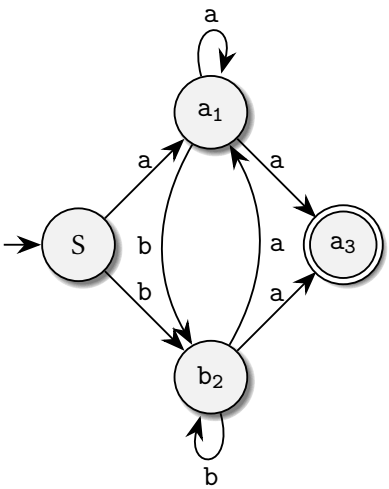

Fig. 1. Glushkov NFA constructed for the regex (a | b)*a.

Definition 5. Given an input alphabet Σ and the memory set cardinality $k \in \mathbb{N}$, a regular expression with backreferences (ref-word) is defined recursively:

- $\gamma \in \Sigma$, ε, and $\&i$, where $i \leq k$, are ref-words (the latter defines containment of the i-th memory cell);
- if ρ_1 and ρ_2 are ref-words, then so are $(\rho_1 \mid \rho_2)$, $(\rho_1\rho_2)$, $(\rho_i)^*$;
- if $i \leq k$ and ρ is a ref-word containing neither $\&i$ nor $[_i\tau]_i$, then $[_i\rho]_i$ is also a ref-word.

The last operation defines capture groups. Thus, unlike expressions in PCRE2 syntax, which permits both unnamed and named capture groups, the ref-words have all their capture groups explicitly named. We require memory brackets $[_i$, $]_i$ to be balanced both wrt the regular parentheses, and wrt each other. That is the only distinction from the formalism given in paper [5], which admits unbalanced capture groups.

The ref-word definition above does not specify semantics of uninitialized backreferences, e.g. in the expression $(\&1\mathtt{a}[_1\mathtt{b}^*]_1)^*$. Following the terminology of the paper [9], we say that we assume *ε-semantics*: all uninitialized references are valued ε.

Proposition 1. *All Kleene algebra identities (i.e. theorems of the form $\forall x_1, \ldots, x_n(\Phi_1 = \Phi_2)$) are true for ref-words.*

Proof. All memory initializations and references inside any instance of Φ_1 and Φ_2 can be replaced with fresh elements of the input alphabet. If the resulting instances of Φ_1 and Φ_2 are equal, then so are their preimages. □

Although the ref-words satisfy most of the Kleene algebra laws, they do not form a Kleene algebra.

Example 1. In the bisimulation law, let us consider $x = [_1\mathtt{a}]_1$, $y = \&1$, $z = \mathtt{aa}$. Then $\mathscr{L}(xy) = \mathscr{L}(yz) = \{\mathtt{aa}\}$, but $\mathscr{L}(x^*y) = \{\varepsilon\} \cup \{\mathtt{a}^{n+2} \mid n \in \mathbb{N}\}$, $\mathscr{L}(yz^*) = \{\mathtt{a}^{2\cdot n} \mid n \in \mathbb{N}\}$.

Thus, the fact that ref-words generate same languages does not imply their equivalence wrt algebraic rewriting rules.

1.4. Memory Finite Automaton

A memory finite automaton (MFA) [5] is a tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$, where Q is a finite set of states, Σ is the input alphabet, $q_0 \in Q$ is a starting state, $F \subseteq Q$ are final states, and $\delta : Q \times (\Sigma \cup \{\varepsilon\} \cup \{\&1, \&2, \ldots, \&k\}) \to$

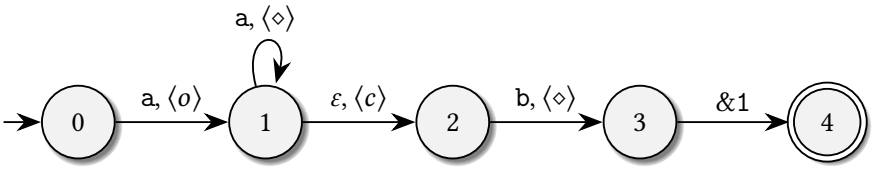

Fig. 2. MFA for $[_1\mathtt{a}^+]_1\mathtt{b}\&1$

$\mathcal{P}(Q \times \{o, c, \diamond\}^k)$ is a transition table. The symbols $o, c, \diamond$ are memory instructions (o — opening, c — closing, $\diamond$ — preserving instruction[9]).

An *MFA configuration* is a tuple $(q, w, (u_1, r_1), \ldots, (u_k, r_k))$, where q is a current state, w is an input to read, and for all $i, 1 \leq i \leq k$, (u_i, r_i) is an i-th memory state (u_i is a stored string; $r_i \in \{O, C\}$ is a memory status, which is either O or C). The initial memory state is $(q_0, w, (\varepsilon, C), \ldots, (\varepsilon, C))$.

A transition from configuration $(q, vw, (u_1, r_1), \ldots, (u_k, r_k))$ to $(p, w, (u_1', r_1'), \ldots, (u_k', r_k'))$ is possible if there is a transition rule $(p, s_1, \ldots, s_k) \in \delta(q, \gamma)$, where either:

- $\gamma \in \Sigma \cup \{\varepsilon\}$ and $v = \gamma$,
- or γ is $\&j$, where $j \in \{1, 2, \ldots, k\}$, and $s_b = c \vee r_j = C$ & $s_j = \diamond$ and $v = u_j$,

and for all i memory states change as follows:

- $(s_i = \diamond)$ & $(r_i = O) \Rightarrow (u_i', r_i') = (u_i v, r_i)$,
- $(s_i = \diamond)$ & $(r_i = C) \Rightarrow (u_i, r_i') = (u_i, r_i)$,
- $s_i = o \Rightarrow (u_i', r_i') = (v, O)$,
- $s_i = c \Rightarrow (u_i, r_i') = (u_i, C)$.

The following rules define how MFA can be constructed from a ref-word (for convenience, the memory instructions are given in a brief form, i.e. the preserving instructions $\diamond$ are omitted).

1. An automaton for a backreference or for a letter is an automaton with the single transition $\delta(q_0, x) = (p), p \in F, x \in \Sigma \cup \{\&1, \&2, \ldots, \&k\}$.
2. If $\langle Q, \Sigma, \delta_\tau, q_0, F\rangle$ is an automaton for a regex τ, then $\langle Q \cup \{q_0'\} \cup \{q_F\}, \Sigma, \delta, q_0', \{q_F\}\rangle$ is the automaton for the regex $[_i\tau]_i$, where the transition function defined as follows:
   - $(p, o_i) \in \delta(q_0', x)$, if $(p) \in \delta_\tau(q_0, x)$;
   - $(q_F, c_i) \in \delta(q, \varepsilon)$, if $(q) \in F$;
   - $(p) \in \delta(q, x)$, if $(p) \in \delta_\tau(q, x)$ $(q \in Q)$.
3. If $\langle Q_1, \Sigma_1, \delta_1, q_{01}, F_1\rangle$ is an MFA for a ref-word τ_1, $\langle Q_2, \Sigma_2, \delta_2, q_{02}, F_2\rangle$ is an MFA for a ref-word τ_2, then $\langle Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta, q_0, F_1 \cup F_2\rangle$ is an MFA for $(\tau_1 \mid \tau_2)$, where $q_0 = q_{01} = q_{02}$, and the transition function is defined as follows:
   - $(p) \in \delta(q, x)$, if $(p) \in \delta_1(q, x) \cup \delta_2(q, x)$.
4. If $\langle Q_1, \Sigma_1, \delta_1, q_{01}, F_1\rangle$ is an MFA for a regex τ_1, $\langle Q_2, \Sigma_2, \delta_2, q_{02}, F_2\rangle$ is an MFA for a regex τ_2, then $\langle Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta, q_{01}, F_2\rangle$ is the MFA for $\tau_1\tau_2$, where δ is defined as follows:
   - $q \in F_1, (p) \in \delta_2(q_{02}, x) \Rightarrow (p) \in \delta(q, x)$;
   - $(p) \in \delta_1(q, x) \Rightarrow (p) \in \delta(q, x)$;
   - $(p) \in \delta_2(q, x)$ & $q \neq q_{02} \Rightarrow (p) \in \delta(q, x)$.
5. If $\langle Q, \Sigma, \delta_\tau, q_0, F\rangle$ is an MFA for a regex τ, then $\langle Q, \Sigma, \delta, q_0, F\rangle$ is the MFA recognizing τ^+ (positive iteration of τ), where δ is specified as follows:
   - $(p) \in \delta_\tau(q, x) \Rightarrow (p) \in \delta(q, x)$;
   - $(p) \in \delta_\tau(q_0, x)$ & $q \in F \Rightarrow (p) \in \delta(q, x)$.

If τ^* is recognized instead of τ^+, then the initial state q_0 is additionally added to the final states of the constructed MFA.

[9] In the more recent paper by Schmid [10], reset instruction is also used. There we omit the resets, because their introduction has no impact on the constructions given in the paper.

An example of a memory finite automaton for a non-regular language $\{\mathtt{a}^{n+1}\mathtt{b}\mathtt{a}^{n+1} \mid n \in \mathbb{N}\}$ is given in Figure 2.

2. Acyclic Regexes and their Disambiguation

Given an academic regex ρ, we can determine ambiguities in ρ, analysing FIRST and FOLLOW sets of its linearised version. In the case of ref-words, we cannot compute FIRST$(\&i)$ without knowing possible values that can be stored in the i-th memory cell. Still, we are only required to track memory initializations that are done last before the memory usage.

Example 2. Given ref-word $\rho_1 = [_1\mathtt{a}^*]_1[_1\mathtt{b}^*]_1\&1$, the memoised block $[_1\mathtt{a}^*]_1$ is useless, because it is re-initialized at any path following it.

Given ref-word $\rho_2 = [_1\mathtt{a}^*]_1([_1\mathtt{b}]_1)^*\&1$, the memoised block $[_1\mathtt{a}^*]_1$ is useful: the memory cell $\&1$ is valued so when the second iteration block (i.e. $([_1\mathtt{b}]_1)^*$) is skipped (no letter b is read from the input string).

Definition 6. An init set $\mathrm{last}_{\mathrm{i:init}}(\rho)$ is the set of all possible values of the i-th memory cell of ρ after reading the whole expression ρ.

A p-positioned init set $\mathrm{last}_{\mathrm{i:init}}(\rho[p])$ is the set of all possible values of the i-th memory cell after reading p-length prefixes of words recognised by ρ.

The values are given in the extended syntax, which makes it possible to express cases when the initialization refers to the other cells. In such a case, e.g. we write $\mathrm{last}_{\mathrm{i:init}}(\rho) = \{[_j\rho']_j\mathtt{b}\&j\}$, in order to point the fact that i-th cell initialization refers to j-th memory, initialized with expression ρ', and then reuses the value of j-th memory cell.

Example 3. Let $\rho = [_1\mathtt{ba}^*]_1[_2\mathtt{ca}^*]_2([_1\&2\mathtt{ab}]_1 \mid [_2\mathtt{bb}^*]_2)^*$.

Then $\mathrm{last}_{2:\mathrm{init}}(\rho) = \{\mathtt{ca}^*, \mathtt{bb}^*\}$. The value $\mathtt{ca}^*$ corresponds to the very first initialization (if no second argument of the alternation under the iteration is used in the parse), and $\mathtt{bb}^*$ corresponds to the case when at least one second argument of the alternation is used in the parse.

The set $\mathrm{last}_{1:\mathrm{init}}(\rho)$ contains three values: $\mathtt{ba}^*$ — by the very first initialization; $[_2\mathtt{ca}^*]_2\mathtt{ab}$ — if a first alternative under the star occurs, without the second alternative preceding it; $[_2\mathtt{bb}^*]_2\mathtt{ab}$ — otherwise.

The previous example demonstrates that the definition of $\mathrm{last}_{\mathrm{i:init}}(\rho)$ is context-sensitive. In fact, it considers traces with all possible memory configurations, and we cannot guarantee that the configurations can be finitely presented as regular expressions, being possibly stacked in references.

The simplest way to restrict the infinite memory dependencies is to detect them syntactically.

Definition 7. Given a ref-word τ, let us say that $i \propto j$ in τ, if $[_iu_1\&ju_2]_i$ occurs in τ as a subword, and u_1 does not contain $]_i$. Let us denote the transitive closure of $\propto$ with $\propto^*$.

We say that a ref-word ρ has a syntactic circularity, if for some $k \in \mathbb{N}$, $\langle k, k\rangle \in \propto^*$.

This criterion is clear and easy to implement. However, if the circularity of the references is defined as the syntactic circularity, some non-circular ref-words can be incorrectly classified as cyclic.

Example 4. Let us consider the ref-word $\tau_1 = ([_1\&2\mathtt{b}]_1\&1[_1\mathtt{a}^*]_1[_2\mathtt{b}^*\&1]_2)^*$. Syntactically, the memory cell 1 depends on 2 recursively (i.e. in a loop) and vice versa, but in fact the real value of the cell 1 is completely reset in every star iteration, and the dependence is finite. Moreover, we can safely rename memory cells in τ_1 in order to obtain a non-cyclic regex $([_3\&2\mathtt{b}]_3\&3[_1\mathtt{a}^*]_1[_2\mathtt{b}^*\&1]_2)^*$.

In ref-word $\tau_2 = ([_1\mathtt{a}^*]_1[_1\&2\mathtt{b}]_1\&1[_2\mathtt{b}^*\&1]_2)^*$ the cyclic dependence occurs, because re-initialized value of the memory cell 1 is not used by the memory cell 2. No equivalent renaming of cells in τ_2 can destroy the loop.

In ref-word $\tau_3 = [_1\mathtt{a}^*]_1\mathtt{b}[_2\&1\&1]_2\mathtt{b}[_1\&2\&2]_1\mathtt{b}\&1$, the cells 1 and 2 depend on each other, but they can again be safely renamed, resulting in the non-cyclic regex $[_1\mathtt{a}^*]_1\mathtt{b}[_2\&1\&1]_2\mathtt{b}[_3\&2\&2]_3\mathtt{b}\&3$.

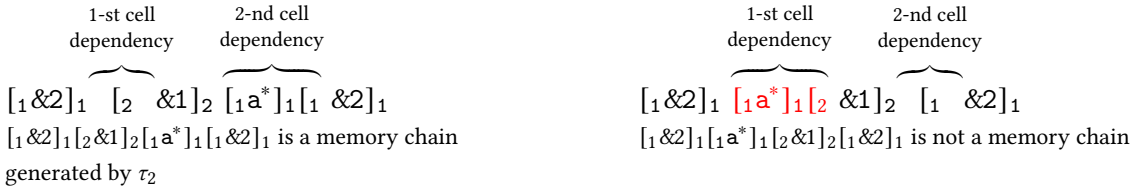

Fig. 3. Memory chains in Example 4

Example 4 points out the following features of ref-word containing real memory loops.

- A memory loop can appear only inside an iteration.
- Inside the iteration, the cells are required to be initialized by references to each other, with no re-initialization of the given cells by the other values until the cyclic dependence occurs.

The second feature can be formulated as follows.

Definition 8. *A dependent memory chain* is a sequence $[_{i_2}\&i_1]_{i_2}[_{i_3}\&i_2]_{i_3}[_{i_n}\&i_{n-1}]_{i_n}$. We say that $\omega \in \Sigma_{\mathcal{M}}{}^*$ *contains* a dependent memory chain if the dependent chain occurs in ω as a scattered subword, and no memory brackets $[_{i_j}]_{i_j}$, $1 < i \leq n$, are used outside of the chain in a subword preceding usage of $\&i_j$ and following the expression $[_{i_j}\&i_{j-1}]_{i_j}$ in the chain.

The memory chains from ref-words in Example 4 are considered in Figure 3.

In order to formulate the loop requirement, we can linearize the memory operations, and require the memory chain to have the same linearization indices of its first and last elements. In fact, an NFA construction is appropriate to track such a dependence.

Definition 9. *Dependency finite automaton* (DepFA) for a ref-word ρ is an NFA for the academic regex $h(\rho)_{\text{lin}}$, where $h(\gamma) = \varepsilon$ for all $\gamma \in \Sigma$, and $h(x) = x$ otherwise.

There the memory brackets (as well as the references) are considered as letters of the input alphabet of the NFA. Thus, the classical Glushkov NFA construction can be used. No nested brackets with same indices are possible, therefore, the factorized expression defines a regular language over linearized version of the alphabet $\Sigma_{\mathcal{M}} = \{\&i, [_i,]_i \mid i \leq k\}$.

Definition 10. A ref-word ρ is *acyclic* if language of its DepFA does not contain a dependent memory chain starting and ending with the same subexpression $[_k\&j_{k'}]_k$. I.e., the linearized indices point out that the dependent memory chain occurs inside an iteration.

We denote the set of acyclic ref-words with **ACREG**.

Thus, the **ACREG** restriction guarantees that no MFA trace has a memory cell depending on its previous value in a loop. Thus, any syntactic circularity can be resolved by cell renaming, if the capture brackets are linearized. Henceforth, we assume that ref-words from **ACREG** do not contain the syntactic circularity. Hence, we can say that the memory cells form a strict partial order wrt to $\propto$ relation, which leads to the following lemma.

Lemma 1. *For any ref-word $\rho \in$ **ACREG** and any i, its set* $\text{last}_{\text{i:init}}(\rho)$ *is finite.*

Proof. For non-iterated subregexes, the property is trivial. Let us consider a leftmost subregex $(\rho_1)^*$ of ρ, thus, $\rho = \rho_0(\rho_1)^*\rho_2$, where for all i $\text{last}_{\text{i:init}}(\rho_0)$ is finite. Since no memory cell can depend on its value, there is at least one index i s.t. for every subregex $[_i\rho']_i$ of ρ_1, ρ' contains no references that are re-initialized in ρ_1. Thus, $\text{last}_{\text{i:init}}(\rho_1{}^*)$ is guaranteed to be finite. Let us call the set of these references $level(0, \rho_1)$

Now given some n, let us choose all the indices of level $n+1$ s.t. their memory cells in ρ_1 depend either on non-re-initialized references or on the references in the set $level(k, \rho_1)$, $k \leq n$. All the sets $\text{last}_{\text{j:init}}(\rho_0(\rho_1)^*)$ of $j \in level(n, \rho_1)$ are finite, thus, the indices of level $n+1$ have finite $\text{last}_{\text{j:init}}$-sets.

Stepwise moving from left to right iterative subregexes in ρ, we can construct only finite sets $\text{last}_{\text{i:init}}(\rho)$ for all i. □

Lemma 1 allows us to introduce the formal definition of the set of last-initialized memories. Below the function $Upd(\mathcal{M}, i, \tau)$ resets the value of i-th memory cell to τ; function $SubstM(\mathcal{M}, \tau)$ substitutes all the first occurrences of non-initialized references $\&i$ in τ by $[_i\mathcal{M}[i]]_i$ (leaving the other references unchanged).

Definition 11. Given a memory cell i and a ref-word τ, the *set of last initializations of i-th memory cell* $\text{last}_{\text{i:init}}(\tau)$ is $\mathcal{M}[i]$, where $\mathcal{M}$ is the set of all memory initializations, defined recursively.

- $SetMem_{\mathcal{M}}(\gamma) := \varnothing$, where γ is a letter or a back-reference;
- $SetMem_{\mathcal{M}}(\tau_1 \mid \tau_2) := SetMem_{\mathcal{M}}(\tau_1) \cup SetMem_{\mathcal{M}}(\tau_2)$;
- $SetMem_{\mathcal{M}}(\tau_1\tau_2) := SetMem_{\mathcal{M}'}(\tau_2)$, where $\mathcal{M}' = SetMem_{\mathcal{M}}(\tau_1)$;
- $SetMem_{\mathcal{M}}([_i\tau]_i) := Upd(\mathcal{M}', i, SubstM(\mathcal{M}', \tau))$, where $\mathcal{M}' = SetMem_{\mathcal{M}}(\tau)$;
- $SetMem_{\mathcal{M}}(\tau^*) := \bigcup SetMem_{\mathcal{M}}(\tau^n)$.

The first two rules of the definition are self-explanatory.

Given a concatenation of two expressions, τ_1 and τ_2, the set of possible memory values after reading both expressions is equal to the set of possible memory values after reading τ_2, updated with respect to memory values changes in τ_1.

Given a capture group i containing an expression τ, the set of memory values of the i-th cell is to be updated with respect to the changes of memory values occurring in τ.

As for the last rule of the definition processing the iteration operation, in absence of memory loops, the set $SetMem_{\mathcal{M}}(\tau^*)$ always stabilizes for some finite value n.

Example 5. Given $\tau = [_1\&2\mathtt{b}]_1\&1[_3\mathtt{a}^*]_3[_2\mathtt{b}\&3]_2$, let us compute its $SetMem_{\langle\varepsilon,\varepsilon,\varepsilon\rangle}(\tau^*)$ value. The memory state $\langle\varepsilon, \varepsilon, \varepsilon\rangle$ comes from ε-semantics used for the ref-words. For simplicity, we represent the whole vector of last initializations by tuples containing alternations, treating the elements of the tuples as sets, and overload the union operation as a mapped union through the tuple elements.

$$SetMem_{\langle\varepsilon,\varepsilon,\varepsilon\rangle}(\tau^*) = \langle\varepsilon, \varepsilon, \varepsilon\rangle \cup SetMem_{\langle\varepsilon,\varepsilon,\varepsilon\rangle}(\tau) \cup SetMem_{\tau}(\tau^{n+1})$$
$$SetMem_{\langle\varepsilon,\varepsilon,\varepsilon\rangle}(\tau) = \langle \mathtt{b}, \mathtt{b}[_3\mathtt{a}^*]_3, \mathtt{a}^*\rangle$$
$$SetMem_{\langle \mathtt{b},\mathtt{b}[_3\mathtt{a}^*]_3,\mathtt{a}^*\rangle}(\tau) = \langle [_2\mathtt{b}[_3\mathtt{a}^*]_3]_2\mathtt{b}, \mathtt{b}[_3\mathtt{a}^*]_3, \mathtt{a}^*\rangle$$

The next star unfolding iteration adds no new values to the memory sets, thus, the resulting memory set is $\langle\varepsilon \mid \mathtt{b} \mid [_2\mathtt{b}[_3\mathtt{a}^*]_3]_2\mathtt{b}, \varepsilon \mid \mathtt{b}[_3\mathtt{a}^*]_3, \varepsilon \mid \mathtt{a}^*\rangle$.

2.1. Ref-words Determinism

If the possible last initializations of the memory cells are known, then we can determine whether a ref-word is unambiguous, looking at the first symbols of the given init-sets. In this subsection, we specify the algorithm computing FIRST set for ref-languages, which is easily used then to track the determinism.

The algorithm for computing FIRST set for academic regular expressions involves the following rule:

$$\text{FIRST}(\rho_1\rho_2) = \begin{cases} \text{FIRST}(\rho_1), \text{ if } \varepsilon \notin \mathscr{L}(\rho_1) \\ \text{FIRST}(\rho_1) \cup \text{FIRST}(\rho_2), \text{ otherwise.} \end{cases}$$

In ref-words, when ρ_1 is valued ε, its value can impose some restrictions on the memory values. Thus, a context-sensitive definition of ε-collapsing is required to track these restrictions.

Definition 12. Collapsing update of backref-normalized regexes is defined as follows:

- $Collapse(\mathcal{M}, \rho_1\rho_2) := Collapse(Collapse(\mathcal{M}, \rho_1), \rho_2)$;
- $Collapse(\mathcal{M}, \&i) := Collapse(Subst(i := \varepsilon, \mathcal{M}), \mathcal{M}[i])$;
- $Collapse(\mathcal{M}, \rho_1 \mid \rho_2) := Collapse(\mathcal{M}, \rho_1) \cup Collapse(\mathcal{M}, \rho_2)$;
- $Collapse(\mathcal{M}, \rho^*) := \mathcal{M}$;

- $Collapse(\mathcal{M}, [_i\rho]_i) := Collapse(Subst(i := \varepsilon, \mathcal{M}), \rho)$;
- $Collapse(\mathcal{M}, \gamma) := \bot$.

Given the fact that every reference has an finite set of initializations, we can formalize the algorithm for computing context-sensitive $\mathrm{FIRST}_{\mathcal{M}}$-set of a ref-word.

Definition 13. Given a ref-word ρ and set of its last initializations $\mathcal{M}$, $\mathrm{FIRST}_{\mathcal{M}}(\rho)$ can be computed recursively:

- $\mathrm{FIRST}_{\bot}(\rho) := \varnothing$
- $\mathrm{FIRST}_{\mathcal{M}}(\gamma) := \{\gamma\}$, where $\gamma \in \Sigma$
- $\mathrm{FIRST}_{\mathcal{M}}(\&i) := \mathrm{FIRST}_{\varnothing}(\mathcal{M}[i])$
- $\mathrm{FIRST}_{\mathcal{M}}([_i\rho]_i) := \mathrm{FIRST}_{\mathcal{M}}(\rho)$
- $\mathrm{FIRST}_{\mathcal{M}}(\rho_1 \mid \rho_2) := \mathrm{FIRST}_{\mathcal{M}}(\rho_1) \cup \mathrm{FIRST}_{\mathcal{M}}(\rho_2)$
- $\mathrm{FIRST}_{\mathcal{M}}(\rho^*) := \mathrm{FIRST}_{SetMem_{\mathcal{M}}(\rho^*)}(\rho)$
- $\mathrm{FIRST}_{\mathcal{M}}(\rho_1\rho_2) := \mathrm{FIRST}_{\mathcal{M}}(\rho_1) \cup \mathrm{FIRST}_{Collapse(\mathcal{M},\rho_1)}(\rho_2)$

Based on the definition, we can formulate the notion of determinism for acyclic ref-words, similarly to the one-unambiguity of academic regular expressions.

Definition 14. Let us say that a ref-word in **ACREG** has a one-ambiguous trace if either:

- it contains a subregex $(\rho_1 \mid \rho_2)\rho_3$ with the memory state $\mathcal{M}$ s.t. $\mathrm{FIRST}_{\mathcal{M}}(\rho_1) \cap \mathrm{FIRST}_{\mathcal{M}}(\rho_2) \neq \varnothing$ or $\mathrm{FIRST}_{Collapse(\mathcal{M},\rho_1)}(\rho_3) \cap \mathrm{FIRST}_{\mathcal{M}}(\rho_2) \neq \varnothing$, or $\mathrm{FIRST}_{Collapse(\mathcal{M},\rho_2)}(\rho_3) \cap \mathrm{FIRST}_{\mathcal{M}}(\rho_3) \neq \varnothing$;
- it contains a subregex $(\rho_1)^*\rho_2$ with the memory state $\mathcal{M}$ s.t. $\mathrm{FIRST}_{SetMem_{\mathcal{M}}(\rho_1{}^*)}(\rho_1) \cap \mathrm{FIRST}_{SetMem_{\mathcal{M}}(\rho_1{}^*)}(\rho_2) \neq \varnothing$.

If a ref-word τ does not contain a one-ambiguous trace, then it is deterministic, i.e. a 1-symbol lookahead can determine the parse path in τ for any input string.

2.2. Ref-words Reversal

We can notice that the referencing operations are in some sense dual to the capture operations, and if all the last initializations are known, we can swap the reading and the writing operations with each other when reading a ref-word from the end. However, if the last initializations are ambiguous, the swap operation cannot be done.

For example, in the ref-word $\rho = [_1\mathtt{ba}^*]_1[_2\mathtt{ca}^*]_2([_1\&2\mathtt{ab}]_1 \mid [_2\mathtt{bb}^*]_2)^*$ the memories are ambiguous, since $SetMem_{\langle\varepsilon,\varepsilon,\varepsilon\rangle}(\rho) = \langle \mathtt{ba}^* \mid [_2\mathtt{ca}^*]_2\mathtt{ab} \mid [_2\mathtt{bb}^*]_2\mathtt{ab}, \mathtt{ca}^* \mid \mathtt{bb}^*\rangle$. So we cannot guess which expression must replace &2 in the reversal ρ^R. On the other hand, if all elements of the memories set of a ref-word have the cardinality 1, then the substitution is uniquely defined.

Definition 15. Let us say that an expression τ is in the semi-backref-normal form (sBNF) if for all its subexpressions τ' ending with $\&i$ the condition $|\mathrm{last}_{\mathrm{i:init}}\tau'| = 1$ holds, and in complete backref-normal form, if additionally every last usage of a reference before its re-initialization or the end of the regex is explicit in τ (i.e. there are no traces where the reference is initialized but not used).

Given $\rho = (\rho_1 \mid \ldots \mid \rho_n)$, where all the ρ_i are in sBNF, but not necessarily have the same $\mathrm{last}_{\mathrm{k:init}}$ sets, we can transform it into the equivalent[10] expression $\rho[_k\varepsilon]_k$, satisfying sBNF condition. Thus, we assume that such ref-words are already in sBNF (and in complete BNF if the corresponding ρ is in BNF).

The two following lemmas verify possibility to transform any element of **ACREG** to backref-normal form. Their proofs are given in the Appendix.

[10] The equivalence holds wrt the language recognized by the expressions, but not wrt the algebraic transformations, hence we always require the given assumption to be implemented only to the whole ref-word, not its subwords.

Lemma 2. *The open-distributivity and nesting transformations preserve semi-backref-normal form of* **ACREG** *elements, and do not increase the cardinalities of the corresponding* $\mathrm{last}_{\mathrm{i:init}}$*-sets.*

Lemma 3. *Every acyclic ref-word* ρ *may be transformed to* ρ' *such that* $\mathscr{L}(\rho) = \mathscr{L}(\rho')$ *and* ρ' *is semi-backref-normal with respect to variable* $\&i$*.*

If a semi-BNF ref-word lacks explicit references preceding re-initializations, it remains to apply the unfolding rule in order to construct the complete BNF.

Example 6. Let us demonstrate the basic steps of backref-normal-form transformation using the ref-word $[_1\mathtt{ba}^*]_1[_2\mathtt{ca}^*]_2([_1\&2\mathtt{ab}]_1 \mid [_2\mathtt{bb}^*]_2)^*$. The red part of the ref-word denotes the subword which is transformed on the current normalization step. First, we remove the capturing operation for the 1-st memory cell, because it is never referenced. Then, we disambiguate the second memory cell.

$$\mathtt{ba}^*[_2\mathtt{ca}^*]_2(\&2\mathtt{ab})^*([_2\mathtt{bb}^*]_2(\&2\mathtt{ab})^*)^* \qquad \text{(nesting rule)}$$

$$\underbrace{(\mathtt{ba}^*[_2\mathtt{ca}^*]_2 \mid \mathtt{ba}^*[_2\mathtt{ca}^*]_2(\&2\mathtt{ab})^*\&2\mathtt{ab})}_{\text{normalised part, denoted by } \bar{}}([_2\mathtt{bb}^*]_2(\&2\mathtt{ab})^*)^* \qquad \text{(unfolding and distributivity)}$$

$$\Psi \mid \Psi\underbrace{([_2\mathtt{bb}^*]_2(\&2\mathtt{ab})^*)^*}_{\text{denoted by } \Phi}[_2\mathtt{bb}^*]_2(\&2\mathtt{ab})^* \qquad \text{(unfolding and distributivity)}$$

$$\Psi \mid \Psi\Phi[_2\mathtt{bb}^*]_2 \mid \Psi\Phi[_2\mathtt{bb}^*]_2(\&2\mathtt{ab})^*\&2\mathtt{ab} \qquad \text{(unfolding and distributivity)}$$

Now every reference to the 2-th memory cell is determined to its value, and all the last references to the memory before the init-set changes are made explicit.

The important property of the BNF ref-words is explicit presence of all last initializations of memory cells in the regex, as well as the explicit presence of all last references to the initializations. Thus, if the memory operations are swapped, and the concatenation arguments are reversed in the academic-regex style, the resulting ref-word will recognize the language of reversed words recognized by the initial ref-word. This property essentially depends on non-circularity of memory, as the following lemma shows.

Lemma 4. *Ref-words languages (without restriction on memory chains) are not closed under reversal.*

Proof. Let us consider regex $\rho = [_1\mathtt{a}]_1\mathtt{b}([_2\&1\&1]_2\mathtt{b}[_1\&2\&2]_1)^*$. Then $\mathscr{L}(\rho) = \{\mathtt{aba}^2\mathtt{b}\ldots\mathtt{ba}^{2^{2k+1}} \mid k \in \mathbb{N}\}$, and it reversal is $\mathscr{L}' = \{\mathtt{a}^{2^{2k+1}}\mathtt{b}\ldots\mathtt{ba} \mid k \in \mathbb{N}\}$. Suppose that $\mathscr{L}'$ is generated by an MFA $\mathscr{A}$, corresponding to backref-regex ρ', containing a finite number of memory cells, say N. Let us consider a derivation of word $\mathtt{a}^{2^{2M+1}}\mathtt{b}\ldots\mathtt{ba}$, where $M = 2^{2^N}$. Then $\mathtt{a}^{2^{2M+1}}$ must be generated by a star fragment in ρ', and ρ' can be represented, possibly after a star unfolding, as $\tau_0(\tau_1)^*\rho''$, where τ_0 and τ_1 contain only letters $\mathtt{a}$ and memorize only the strings in the alphabet $\{\mathtt{a}\}$. Note that $\mathtt{a}^{2^{2M-1}}\mathtt{b}\ldots\mathtt{ba}$ contains 2^{2^N} letters $\mathtt{b}$, thus, they are also generated by a star fragment in ρ''. Thus, ρ'' can be rewritten as $\tau_2(\tau_3)^*\tau_4$. The words generated by $\tau_0(\tau_1)^*\tau_2\tau_4$ (collapsing $(\tau_3)^*$ block to ε) also belong to $\mathscr{L}'$, but they can contain only $O(2^N)$ letters $\mathtt{b}$, while the words starting with $\mathtt{a}^{2^{2M+1}}$ must contain at least 2^{2^N} letters $\mathtt{b}$. Hence, $\mathscr{L}'$ does not belong to MFA languages. □

Most **ACREG** ref-words are easily reversed after the BNF construction. The main complication of the ref-word reversal in **ACREG** class occurs when a ref-word contains a memory chain inside an iteration, where the dependent cells are in mixed "read-write" form, i.e. the references to them precede their re-initializations. Then the BNF itself becomes insufficient to give the reversal construction, and the sliding and fusion rewriting rules are to be implemented in order to link the initializations and the references occurring in the different iterations of the same subexpression.

Example 7. Given a ref-word $\rho = [_1\mathtt{a}^*]_1(\&2\mathtt{b}[_2\&1\mathtt{c}^*]_2[_1\mathtt{a}^*]_1)^*$, the ref-word ρ^R requires three memory cells.

Indeed, let us unfold the outer star operation twice and specify the iterations by parameters:

$$\mathtt{a}^{k_0}\mathtt{b}\mathtt{a}^{k_0}\mathtt{c}^{k_1}\mathtt{a}^{k_2}\mathtt{a}^{k_0}\mathtt{c}^{k_1}\mathtt{b}\mathtt{a}^{k_2}\mathtt{c}^{k_3}\mathtt{a}^{k_4}.$$

The subexpression $\mathtt{a}^{k_0}$ is used between occurrences of $\mathtt{a}^{k_2}$ and of $\mathtt{c}^{k_1}$, and it is used by itself, thus, it requires an independent memory cell in the reversal ref-word. Similarly, the memories containing $\mathtt{a}^{k_2}$ and $\mathtt{c}^{k_1}$ must be separated from each other.

The reversal actually can be constructed, if we use three memory cells instead of two, and sliding together with fusion: $\tau^* = (\tau\tau)^*(\tau \mid \varepsilon)$. Now we only give the construction of the reversed ref-word, not specifying the complete algorithm.

$$\mathtt{a}^*\mathtt{c}^*([_1\mathtt{a}^*]_1 \mid [_3\mathtt{a}^*]_3[_2\mathtt{c}^*[_1\mathtt{a}^*]_1]_2\&3\&2)(\mathtt{b}[_2\mathtt{c}^*[_3\mathtt{a}^*]_3]_2\&1\&2\mathtt{b}[_2\mathtt{c}^*[_1\mathtt{a}^*]_1]_2\&3\&2)^*\mathtt{b}\mathtt{c}^*\&1.$$

While the "read–write" dependent memory cells yield critical growth in the reversed ref-word size, they are mostly of the theoretical interest. We found such regexes neither in the StackOverflow nor RegexLib data sets.

3. Experiments

The backref-normal form and ref-word reversal algorithms were implemented in a model regex engine that only supports backreferences and basic academic regex syntax. We implemented ref-word-to-MFA transformation designed by Schmid [5], and matching method based on Thompson's algorithm extended to MFA. The algorithms implementing the base transformations of ref-words are given in the corresponding section of Appendix.

In order to validate our formalization assumptions, we collected a set of regexes from StackOverflow[11] and RegexLib[12] and filtered out only ones containing backreferences. Analysing these regexes led to the following observations:

- Cyclic memory dependency does not occur in real regexes. Therefore, the **ACREG** formalization is sufficient for practical purposes.
- Dependent memory chains are rare (only approximately 60 out of more than 3000 regexes with backreferences), and none of them have the "read–write" structure, requiring fusion transformation. Hence, memory cells explosion in reversed regexes is unlikely to occur.

As a test base, we collected a set of regexes with backreferences that can be presented in Python syntax, in order to compare the model implementation with the practical extended regex engine. That condition imposed some restrictions on the regexes form, since Python syntax does not support re-defining capture groups. Memory circularity is still possible in it, e.g. in $(\mathtt{a}^*\mathtt{b} \mid \backslash 1)^*$. Most of the regexes were taken from the StackOverflow dataset, sometimes with syntax simplification to meet the requirements of the model implementation. Each regex was considered in four forms: the initial one, backref-normalized, reversed, and in Python syntax. In order to track matching time growth, we used the "malicious pump" technique [11]: the regex was matched against the strings $s_1 s_2^k s_3$, where $k \in \mathbb{N}$ simultaneously increases until the timeout occurs.

The experiments demonstrated that none of the transformed regexes exhibited superlinear slowdown in our implementation compared to the Python regex engine. Thus, the overheads of using longer ref-words (after the nesting and unfolding transformations) were offset by reduced backtracking, due to limited non-determinism in the memory of resulting ref-words.

[11] https://github.com/SBULeeLab/LinguaFranca-FSE19
[12] https://regexlib.com

Reversed and normalized regexes also allow for more matching heuristics since every reference in such a regex is necessarily used after the corresponding initialization. In the MFA matching algorithm, this means that we can prune matching paths when the remaining input string is not long enough to match all words that have been written to the memory but have not been read yet.

Overall, reversed regexes and initial regexes showed comparable matching time growth in most cases (58 % in total). By saying "comparable matching time growth", we mean that non-reversed and reversed regexes reach timeout (0.5 sec) on input strings of lengths that differ from each other by no more than one order of magnitude. In 25 % of cases, the reversed regexes were significantly more efficient (timeout strings lengths differ by more than one order of magnitude), while in the remaining 17 % of cases, the non-reversed regexes were superior in terms of matching time. More detailed results are displayed in Figure 4, classifying regexes by difference in orders of magnitude of their timeout strings.

We sorted regexes by their iteration structure, identifying 4 groups:

1. Regexes with iteration in a capture group.
2. Regexes with iteration over a backreference.
3. Regexes with iteration in a capture group and over a backreference.
4. RW regexes — may produce a match where a cell references to a previous iteration step.

The performance benefits of using reversed regexes can be attributed to two main reasons. Firstly, a regex may be less ambiguous when matched from the right. Secondly, in cases where left and right ambiguities are comparable, the matching time reduces by means of described optimization for particular input strings that cause memoization of long words. However, it is important to note that reversing a regex can also have a negative impact on performance — if a regex is more ambiguous when matched from the right, the reversed ref-word MFA contains more states. Another feature that affects performance is the backref normalization: all transformations increase the length of the regex that results in large amount of states which offset any potential benefits of reversal. Given the one-unambiguity criterion presented in Section 2, we can determine whether the reversed and non-reversed regexes generate deterministic MFAs, but we still cannot distinguish between slightly and heavily non-deterministic regexes.

4. Related works

Multiple formalizations of regular expressions with backreferences have been proposed in the literature. One such formalization, suggested by Campeanu, Salomaa, and Yu [12], is based on PCRE2 syntax and involves implicit numbering of capture groups. Thus, memory cells cannot be reinitialized, and every backreference must be preceded by the corresponding closed capture group, which completely forbids cyclic memory dependencies. Campeanu–Salomaa–Yu backref-languages de-facto describe languages of the most practical backref-regexes, and satisfy desirable properties like the extended pumping lemma. However, the fact that the capture groups are anonymous ruins algebraic properties of this class, making it impossible to use equivalence rules for regex rewriting.

In 2014, Markus Schmid suggested a new formalism for the capture groups: all the groups are named and their structure is independent of the core regex structure [5]. Thus, unbalanced memory brackets across different memory cells are allowed, enabling ref-words like $[_1\mathtt{a}^*[_2\mathtt{b}^*]_1\mathtt{a}]_2$. Model implementations of the Schmid-style MFA matching algorithms have shown to be efficient compared to backtracking-based matching algorithms. While Schmid formalization is convenient for transforming regular expressions to memory finite automata (also introduced in [5]), the unbalanced structures cannot be represented as tree terms, making rewriting rules difficult to apply. Schmid-style regular expressions are more expressible than PCRE2-style regular expressions[13] used in practice [9], which results in complications in the algorithms rewriting them. In the paper [13], authors attempted to formalise an analogue of the uniqueness of last initialisations

[13]The statement holds if the only non-regular operation considered is back-referencing a string; lookaheads, lookbehinds and, especially, recursive capture groups can be used to express languages non-expressible even in Schmid formalism.

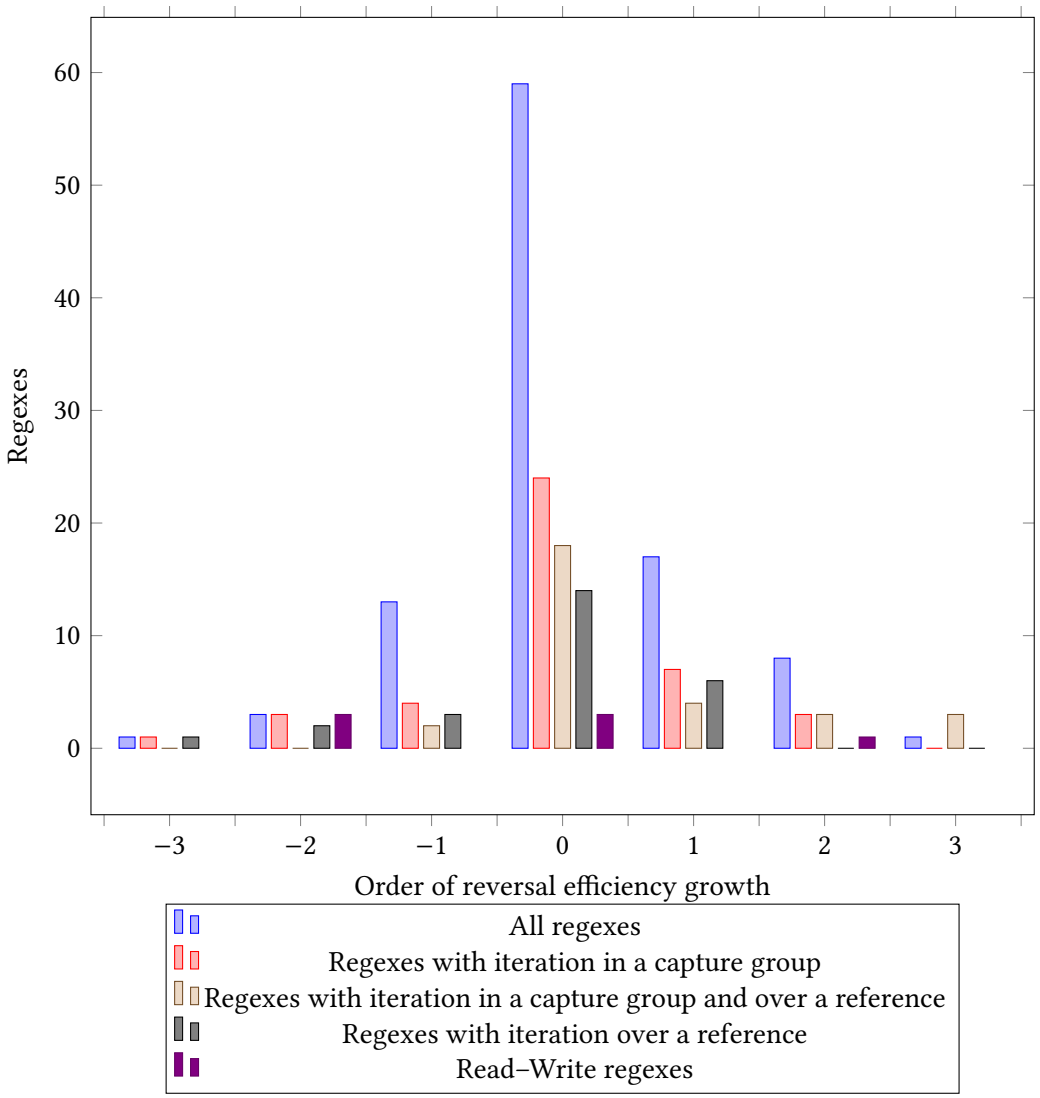

Fig. 4. Comparison of reversed and initial regexes in the MFA implementation

given in a regex, but the star case is not treated consistently in this formalisation (a possibility of an empty iteration is ignored).

In the paper [10], Schmid introduced a polynomial matching time algorithm for a subclass of MFAs with the bounded reference distance. His algorithm leverages the similarity between ref-words (i.e. extended regexes) and patterns, and relies on the paper [14], describing efficient matching algorithms for patterns with bounded variable distance. Moreover, the paper [15] introduces a formalism for recursive regular patterns, which brings pattern languages and backref-languages closer together. It is easy to notice that the recursive patterns can be efficiently reversed, if the variables on layers are not mixed. However, the recursive regex formalization is hardly compatible with the iteration unfolding rule. If there are independently introduced variables under a Kleene star, each unfolding must introduce fresh variables, e.g. $(\mathtt{XaX})^*$, where $\mathtt{X}$ is a pattern variable, is unfolded to $\mathtt{X}_1\mathtt{a}\,\mathtt{X}_1\mathtt{X}_2\mathtt{a}\,\mathtt{X}_2 \ldots \mathtt{X}_n\mathtt{a}\,\mathtt{X}_n$.

Algebraic rewriting is a widely applied technique for academic regular expressions to improve matching efficiency. The strong star-normal form mentioned earlier [16] is a simple example of this technique. A more complex example is given in paper [17], where the whole expressiveness of Kleene algebra is used to reduce size of academic regular expressions. Due to PSPACE-hardness of equivalence in Kleene algebras, the paper

uses bounded-size optimizations. Paper [18] describes experience of using rewriting rules to heuristically optimize academic regexes and avoid catastrophic backtracking. In papers [19, 20] pattern-based rewriting approach has also been used to locally repair suspicious subregexes using heuristically derived rewriting rules.

Conclusion

The semantic-preserving rewriting approach showed itself as a promising way to boost efficiency of context-sensitive extensions of regular expressions. The restrictions imposed on the regex set (namely, the non-circularity) allowed us to define the reversal algorithm and partly adopt the RE2 flexible regex matching algorithm for the regular expressions with backreferences. However, a lot of research remains to be a future work.

- The dependent-memory checking algorithm requires more accurate implementation. For now, it utilizes properties of finite automata, but the finite automata over the linearised words are known to be hard to analyse in general (for example, they can define languages close to Shutzenberger one, imposing worst-case bounds on automata intersection or complementation [2]).
- Real-world regexes are not only checked for the full matching, but also are wanted to preserve the substitutions in given capture groups. For example, a standard definition of Kleene star is greedy, i.e. every iteration tries to capture a longest possible string. Thus, in the reversed regex, the matching groups must become lazy, in order to respect the substitutions.
- Subtleties with the read-write dependent memory chains make the reversal algorithm for the whole class of **ACREG** ref-words almost intractable. It would be interesting to implement a certified version of the algorithm, at least for non-read-write regexes.
- Since the rewriting rules in BNF construction are used only in the length-increasing direction, the worst-case and average-case estimations of regex growth are required.
- The last but not the least: in contrast from the academic case, lookaheads are not a mere syntactic sugar in the case of extended syntax, they extend expressiveness of the languages [13, 21]. We observed that the lookaheads are frequently used with the backreferences in the Internet regexes. Thus, studying this extension would be also of a practical value.

Acknowledgements

We give special thanks to BMSTU student A. Zadvornykh, who helped to classify the StackOverflow regexes wrt reference structure, and generally to BMSTU students from Theoretical Computer Science department whose passion to knowledge inspired the authors.

References

[1] K. R. Beesley, “Kleene, a free and open-source language for finite-state programming”, in *Finite-State Methods and Natural Language Processing*, 2012, pp. 50–54.

[2] W. Gelade and F. Neven, “Succinctness of the complement and intersection of regular expressions”, *ACM Transactions on Computational Logic*, vol. 13, no. 1, 2012. DOI: 10.1145/2071368.2071372.

[3] V. M. Glushkov, “The abstract theory of automata”, *Russian Mathematical Surveys*, vol. 16, no. 5, pp. 3–62, 1961, in Russian.

[4] D. Angluin, “Finding patterns common to a set of strings”, *Journal of Computer and System Sciences*, vol. 21, no. 1, pp. 46–62, 1980. DOI: 10.1016/0022-0000(80)90041-0.

[5] M. L. Schmid, “Characterising REGEX languages by regular languages equipped with factor-referencing”, *Information and Computation*, vol. 249, pp. 1–17, 2016. DOI: 10.1016/j.ic.2016.02.003.

[6] J. Goodman, “Semiring parsing”, *Computational Linguistics*, vol. 25, pp. 573–605, 1999.

[7] D. Kozen, "A completeness theorem for kleene algebras and the algebra of regular events", *Information and Computation*, vol. 110, no. 2, pp. 366–390, 1994. DOI: 10.1006/inco.1994.1037.

[8] A. Bruggemann-Klein and D. Wood, "One-unambiguous regular languages", *Information and Computation*, vol. 140, no. 2, pp. 229–253, 1998. DOI: 10.1006/inco.1997.2688.

[9] M. Berglund and B. van der Merwe, "Re-examining regular expressions with backreferences", *Theoretical Computer Science*, vol. 940, pp. 66–80, 2023. DOI: 10.1016/j.tcs.2022.10.041.

[10] D. D. Freydenberger and M. L. Schmid, "Deterministic regular expressions with back-references", *Journal of Computer and System Sciences*, vol. 105, pp. 1–39, 2019. DOI: 10.1016/j.jcss.2019.04.001.

[11] Y. Li *et al.*, "ReDoSHunter: A combined static and dynamic approach for regular expression DoS detection", in *30th USENIX Security Symposium*, 2021, pp. 3847–3864.

[12] C. Campeanu, K. Salomaa, and S. Yu, "A formal study of practical regular expressions", *International Journal of Foundations of Computer Science*, vol. 14, pp. 1007–1018, 2003. DOI: 10 . 1142 / S012905410300214X.

[13] N. Chida and T. Terauchi, "On lookaheads in regular expressions with backreferences", *IEICE Transactions on Information and Systems*, vol. E106–D, no. 5, pp. 959–975, 2023. DOI: 10.1587/transinf.2022EDP7098.

[14] D. Reidenbach and M. L. Schmid, "Patterns with bounded treewidth", *Information and Computation*, vol. 239, pp. 87–99, 2014. DOI: 10.1016/j.ic.2014.08.010.

[15] M. L. Schmid, "Inside the Class of REGEX Languages", in *Proceedings of the 16th International Conference on Developments in Language Theory*, 2012, pp. 73–84. DOI: 10.1007/978-3-642-31653-1_8.

[16] A. Brüggemann-Klein, "Regular expressions into finite automata", *Theoretical Computer Science*, vol. 120, no. 2, pp. 197–213, 1993. DOI: 10.1016/0304-3975(93)90287-4.

[17] S. Kahrs and C. Runciman, "Simplifying regular expressions further", *Journal of Symbolic Computation*, vol. 109, pp. 124–143, 2022. DOI: 10.1016/j.jsc.2021.08.003.

[18] J. McClurg, M. Claver, J. Garner, J. Vossen, J. Schmerge, and M. E. Belviranli, "Optimizing regular expressions via rewrite-guided synthesis", in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2023, pp. 426–438. DOI: 10.1145/3559009.3569664.

[19] Y. Li *et al.*, "RegexScalpel: Regular expression denial of service (ReDoS) defense by Localize-and-Fix", in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4183–4200.

[20] N. Chida and T. Terauchi, "Repairing DoS vulnerability of real-world regexes", in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 2060–2077. DOI: 10.1109/SP46214.2022.9833597.

[21] Y. Uezato, "Regular Expressions with Backreferences and Lookaheads Capture NLOG", in *51st International Colloquium on Automata, Languages, and Programming (ICALP 2024)*, 2024, 155:1–155:20. DOI: 10.4230/LIPIcs.ICALP.2024.155.

Appendix

4.1. Proof of Lemma 2

- Let ρ be $\rho_1(\rho_2 \mid \rho_3)\rho_4$ in semi-backref-normal form. After opening of concatenation, the regex becomes $\rho' = (\rho_1\rho_2\rho_4 \mid \rho_1\rho_3\rho_4)$, and includes no new alternating subregexes, as compared to ρ. Thus, its ambiguity can only decrease.
- Let ρ be $\rho_1(\rho_2 \mid \rho_3)^*\rho_4$ in sBNF and $\rho' = \rho_1{\rho_2}^*({\rho_3\rho_2}^*)^*\rho_4$ be its image after the nesting transformation. Since both regexes are acyclic, their $\text{last}_{\text{i:init}}$ sets are computed for the bounded depth of iterations

of the Kleene star. But in both cases, the bounded iterations of $(\rho_3\rho_2{}^*)$ coincide with the bounded iterations of $(\rho_2 \mid \rho_3)$, with the unfolding and open-distributivity laws applied. Note that the unfolding transformation does not change the regex $\text{last}_{\text{i:init}}$ set: it only rewrites the star operator corresponding to its definition. Hence, due to open distributivity transformations, the ambiguity of the nested expression can either be preserved, or decreased, as compared to ρ.

4.2. Proof of Lemma 3

The following algorithm decreases the cardinality of the set $\text{last}_{\text{i:init}}$ for every backreference in ρ.

- Consider $\rho = (\rho_1 \mid \ldots \mid \rho_n)\rho_{n+1}$ where ρ_{n+1} contains $\&i$ and does not contain $[_i s]_i$, and $\exists k_1, k_2 \leq n : \text{last}_{\text{i:init}}(\rho_{k_1}) \neq \text{last}_{\text{i:init}}(\rho_{k_2})$. We apply distributivity to ρ, getting $\rho^{'} = (\rho_{1,1} \mid \ldots \mid \rho_{1,j_1})\rho_{n+1} \mid \ldots (\rho_{m,1} \mid \ldots \mid \rho_{m,j_m})\rho_{n+1}$ where $\forall s, k, k^{'} : \text{last}_{\text{i:init}} r_{s,k} = \text{last}_{\text{i:init}} r_{s,k'}$.
- Consider $\rho_0{\rho_1}^*\rho_2$ where ρ_2 contains $\&i$ but not $[_i s]_i$ and $\text{last}_{\text{i:init}}(\rho_0) \neq \text{last}_{\text{i:init}}(\rho_1)$, while ρ_1 has unavoidable initializations of i-th memory cell. Then the regex $(\rho_0\rho_2 \mid \rho_0{\rho_1}^*\rho_1^k\rho_2)$ has at least one less element in the $\text{last}_{\text{i:init}}$ set preceding the given $\&i$.
- Consider $(\rho_1 \mid \rho_2)^*\rho_3$ where ρ_3 contains $\&i$ but not $[_i s]_i$, and $\text{last}_{\text{i:init}}(\rho_1) \neq \text{last}_{\text{i:init}}(\rho_2)$. Its image after the nesting transformation is $\rho^{'} = (\rho_1{}^*\rho_2)^*\rho_1{}^*\rho_3$. If we apply the unfolding transformation to the star operators in ρ', we can make the traces with the last initialization from ρ_1 and the traces with the last initialization from ρ_2 explicit, avoiding the ambiguity of the references in ρ_3.

Iteration and alternation are the only sources of ambiguity, hence by applying above-described transformation we can reduce cardinality of all $\text{last}_{\text{i:init}}$ sets to 1 after finite number of steps, by Lemma 1.

5. Pseudocodes

5.1. MFA construction

```
function MFA(bt)
    switch bt.type do
        case epsilon, literal, backreference
            Q := {v, u}; q_0 := v; Q_finish := u; v.to = {u}
            v.transitions[u] := bt.value
            return Q
        case initialization
            Q := MFA(bt.child)
            for v ∈ q_0.to do
                q_0.op[v] = ⟨x, O⟩
            for v ∈ Q, v.to ∩ Q_final ≠ ∅ do
                for u ∈ v.to ∩ Q_final do
                    v.op[u] := ⟨x, C⟩
            return Q
        case alternation
            Q_left := MFA(bt.left); Q_right := MFA(bt.right)
            Q = {s}; s.to := q_0,left.to ∪ q_0,right.to
            s.transitions := q_0,left.transitions ∪ q_0,right.transitions
            for v ∈ Q_left ∪ Q_right \ {q_0,left, q_0,right} do
                Q := Q ∪ {v}
            return Q
        case concatenation
            Q_left := MFA(bt.left); Q_right := MFA(bt.right)
```

```
        Q = ∅
        for v ∈ Q_left ∪ Q_right do
            if v ≠ q_{0,right} then
                Q := Q ∪ v
            if v ∈ Q_{left,finish} then
                v.transitions := q_{0,right}.transitions
        return Q
    case kleenePlus
        Q = MFA(bt.child)
        for {v, u} ∈ Q do
            if v ∈ q_0.to, u.to ∩ Q_final ≠ ∅ then
                for t ∈ u.to ∩ Q_final do
                    u.to := u.to ∪ {v}
                    u.transitions[v] := Concat(u.transitions[f], q_0.transitions[v])
        return Q
```

5.2. Ref-word Reversal

```
function reverse(regexp)
    switch regexp.type do
        case epsilon, literal, backreference
            return regexp
        case kleeneStar, kleenePlus
            regexp.subregexp = regexp.subregexp._reverse
        case concatenation
            for subregexp in regexp.subregexps do
                newregexp.subregexps.push_front(subregexp._reverse)
        case alternation
            for subregexp in regexp.subregexps do
                newregexp.subregexps.push_back(subregexp._reverse)
```

5.3. Swapping Memory Operations

```
function replace_read_write(r, initialized)
    switch regexp.type do
        case epsilon, literal return
        case backreference
            if x ∈ initialized then return
            else
                r := replace_read_write(r.ref_to, initialized)
                initialized := initialized ∪ {x}
        case initialization
            if x ∈ initialized then
                r := backreference(x)
            else
                initialized := initialized ∪ {x}
                return
        case kleeneStar, kleenePlus
```

```
            replace_read_write(r.subregexp, initialized)
        case concatenation, alternation
            for s ∈ r.subregexps do
                replace_read_write(s, initialized)
```

5.4. Backref-Normal-Form algorithm

```
function BNF(r)
    switch r.type do
        case epsilon, literal, backreference
            return
        case alternation. concatenation
            for s ∈ r.subregexps do
                BNF(s)
            if r.typeis concatenation then
                for s ∈ r.subregexps do
                    for ref ∈ r.refs do
                        if lastinitAmb(ref) then
                            for init ∈ lastinit(ref), lastinitAmb(ref) do
                                if underKleeneStar(init) then
                                    Sliding(init)
                                if underAlternation(init) then
                                    Distribute(init)
        case kleeneStar, kleenePlus
            BNF(r.subregexp)
            if IsAlternationRW(r.subregexp) then
                A := RWOptions(r.subregexp)
                B := AllOptions(r.subregexp) \ A
                r := Denesting(A, B)
            if IsConcatenationRW(r.subregexp) then
                A := ReadSubregexp(r.subregexp)
                B := WriteSubregexp(r.subregexp)
                Denesting(A, B)
                if lastinitAmb(A) then
                    Sliding(r)
            if IsAlternation(r.subregexp) then
                A := memoryFreeOptions(r.subregexp) ∪ UninitializedRefsOptions(r.subregexp)
                B := AllOptions(r.subregexp) \ A
                Denesting(A, B)
        case initialization
            BNF(r.subregexp)
    clear(regexp)                    ▷ Clear uninitialized references and unused initializations
```