Deterministic Regular Expressions With Back-References

Dominik D. Freydenberger[*1] **and Markus L. Schmid**[2]

1 **University of Bayreuth, Germany,** ddfy@ddfy.de
2 **University of Trier, Germany,** MSchmid@uni-trier.de

Abstract

Most modern libraries for regular expression matching allow back-references (i. e., repetition operators) that substantially increase expressive power, but also lead to intractability. In order to find a better balance between expressiveness and tractability, we combine these with the notion of determinism for regular expressions used in XML DTDs and XML Schema. This includes the definition of a suitable automaton model, and a generalization of the Glushkov construction.

1998 ACM Subject Classification F.1.1 Models of Computation, F.4.3 Formal Languages

Keywords and phrases Deterministic Regular Expression, Regex, Glushkov Automaton

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Regular expressions were introduced in 1956 by Kleene [34] and quickly found wide use in both theoretical and applied computer science. While the theoretical interpretation of regular expressions remains mostly unchanged (as expressions that describe exactly the class of regular languages), modern applications use variants that vary greatly in expressive power and algorithmic properties. This paper tries to find common ground between two of these variants with opposing approaches to the balance between expressive power and tractability.

The first variant that we consider are *regex*, regular expressions that are extended with a *back-reference operator*. This operator is used in almost all modern programming languages (like e. g. Java, PERL, and .NET). For example, the regex $\langle x\colon (\mathtt{a} \vee \mathtt{b})^*\rangle \cdot \&x$ defines $\{ww \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$, as $(\mathtt{a} \vee \mathtt{b})^*$ can create a $w \in \{\mathtt{a},\mathtt{b}\}^*$, which is then stored in the variable x and repeated with the reference $\&x$. Hence, back-references allow to define non-regular languages; but with the side effect that the membership problem is NP-complete (cf. Aho [2]).

The other variant, *deterministic regular expressions* (also known as *1-unambiguous regular expressions*), uses an opposite approach, and achieves a more efficient membership problem than regular expressions by defining only a strict subclass of the regular languages.

Intuitively, a regular expression is deterministic if, when matching a word from left to right with no lookahead, it is always clear where in the expression the next symbol must be matched. This property has a characterization via the *Glushkov construction* that converts every regular expression α into a (potentially non-deterministic) finite automaton $\mathcal{M}(\alpha)$, by treating each terminal position in α as a state. Then α is deterministic if $\mathcal{M}(\alpha)$ is deterministic. As a consequence, the membership problem for deterministic regular expressions can be solved more efficiently than for regular expressions in general (more details can be found in [31]). Hence, in spite of their limited expressive power, deterministic regular expressions are used

* Dominik D. Freydenberger was supported by DFG grant FR 3551/1-1.

in actual applications: Originally defined for the ISO standard for SGML (see Brüggemann-Klein and Wood [9]), they are a central part of the W3C recommendations on XML DTDs [7] and XML Schema [27] (see Murata et al. [41]).

The goal of this paper is finding common ground between these two variants, by introducing *deterministic regex* and an appropriate automaton model, the *deterministic memory automata with trap-state* (DTMFA). To elaborate: We first introduce a new automaton model for regex, the memory automata with trap-state (TMFA). While the TMFA is based on the MFA that was proposed by Schmid [45], its deterministic variant, the DTMFA, is better suited for complementation than the deterministic MFA. We then generalize the notion of deterministic regular expressions to regex, and show that the Glushkov construction can also be generalized. This allows us not only to efficiently decide the membership problem for deterministic regex, but also whether a regex is deterministic. After this, we study the expressive power of these models. Although deterministic regex share many of the limitations of deterministic regular expressions (in particular, the inherent non-determinism of some regular languages persists), their expressive power offers some surprises. Finally, we examine a subclass of deterministic regexes and DTMFA for which polynomial space minimization is possible, and we consider an alternative notion of determinism.

From the perspective of deterministic regular expressions, this paper proposes a natural extension that significantly increases the expressive power, while still having a tractable membership problem. From a regex point of view, we restrict regex to their deterministic core, thus obtaining a tractable subclass. Hence, the authors intend this paper as a starting point for further work, as it opens a new direction on research into making regex tractable. For space reasons, detailed proofs are given in a full version of the paper [26].

Main contributions. The main conceptual contribution of this paper are the notion of determinism in regex, and an appropriate deterministic automaton model. The main challenge from this point of view was finding a natural extension of deterministic regular expressions that preserves the following properties: A natural definition of determinism that can be checked efficiently and also has an automata-theoretic characterization, and an efficient Glushkov-style conversion to automata that decide the membership problem efficiently. Regarding technical contributions, the authors would like to emphasize that, in addition to the effort that was needed to accomplish the aforementioned goals, the paper uses subtleties of the back-reference operator in novel ways. By using these, deterministic regex can define non-deterministic regular languages (in particular, all unary regular languages), as well as infinite languages that are not pumpable in the usual sense.

Related work. Regex were first examined from a theoretical point of view by Aho [2], but without fully defining the semantics. There were various proposals for semantics, of which we mention the first by Câmpeanu, Salomaa, Yu [10], and the recent one by Schmid [45], which is the basis for this paper. Apart from defining the semantics, there was work on the expressive power [10, 11, 25], the static analysis [11, 23, 24], and the tractability of the membership problem (investigated in terms of a strongly restricted subclass of regex) [21, 22]. They have also been compared to related models in database theory, e. g. graph databases [4] and information extraction [20, 24].

Following the original paper by Brüggemann-Klein and Wood [9], deterministic regular expressions have been studied extensively. Aspects include computing the Glushkov automaton and deciding the membership problem (e. g. [8, 31, 43]), static analysis (cf. [40]), deciding whether a regular language is deterministic (e. g. [16, 31, 39]), closure properties and descriptional complexity [37], and learning (e. g. [5]). One noteworthy extension are counter operators (e. g. [29, 31, 36]), which we briefly address in Section 7.

2 Preliminaries

We use ε to denote the *empty word*. The subset and proper subset relation are denoted by $\subseteq$ and $\subset$, respectively. Let Σ be a finite terminal alphabet. Unless otherwise noted, we assume $|\Sigma| \geq 2$. Let Ξ be an infinite variable alphabet with $\Xi \cap \Sigma = \emptyset$. Let $w \in \Sigma^*$, then, for every i, $1 \leq i \leq |w|$, $w[i]$ denotes the symbol at position i of w. We define $w^0 := \varepsilon$ and $w^{i+1} := w^i \cdot w$ for all $i \geq 0$, and, for $w = a_1 \cdots a_n$ with $a_i \in \Sigma$, let $w^{m+\frac{i}{n}} = w^m \cdot a_1 \cdots a_i$ for all $m \geq 0$ and all i with $0 \leq i \leq n$. A $v \in \Sigma^*$ is a *factor* of w if there exist $u_1, u_2 \in \Sigma^*$ with $w = u_1 v u_2$. If $u_2 = \varepsilon$, v is also a *prefix* of w.

We use the notions of deterministic and non-deterministic finite automata (DFA and NFA) like [32]. If an NFA can have ε-transitions, we call it an ε-NFA. Given a class $\mathcal{C}$ of language description mechanisms (e. g., a class of automata or regular expressions), we use $\mathcal{L}(\mathcal{C})$ to denote the class of all languages $\mathcal{L}(C)$ with $C \in \mathcal{C}$. The *membership problem for* $\mathcal{C}$ is defined as follows: Given a $C \in \mathcal{C}$ and a $w \in \Sigma^*$, is $w \in \mathcal{L}(C)$?

2.1 Regex

▶ **Definition 1** (Syntax of regex)**.** We define RX, the set of *regex* over Σ and Ξ, recursively:
Terminals and ε: $a \in \mathsf{RX}$ and $\mathsf{var}(a) = \emptyset$ for every $a \in (\Sigma \cup \{\varepsilon\})$.
Variable reference: $\&x \in \mathsf{RX}$ and $\mathsf{var}(\&x) = \{x\}$ for every $x \in \Xi$.
Concatenation: $(\alpha \cdot \beta) \in \mathsf{RX}$ and $\mathsf{var}(\alpha \cdot \beta) = \mathsf{var}(\alpha) \cup \mathsf{var}(\beta)$ if $\alpha, \beta \in \mathsf{RX}$.
Disjunction: $(\alpha \vee \beta) \in \mathsf{RX}$ and $\mathsf{var}(\alpha \vee \beta) = \mathsf{var}(\alpha) \cup \mathsf{var}(\beta)$ if $\alpha, \beta \in \mathsf{RX}$.
Kleene plus: $(\alpha^+) \in \mathsf{RX}$ and $\mathsf{var}(\alpha^+) = \mathsf{var}(\alpha)$ if $\alpha \in \mathsf{RX}$.
Variable binding: $\langle x \colon \alpha \rangle \in \mathsf{RX}$ and $\mathsf{var}(\langle x \colon \alpha \rangle) = \mathsf{var}(\alpha) \cup \{x\}$ if $\alpha \in \mathsf{RX}$ with $x \in \Xi \setminus \mathsf{var}(\alpha)$.
In addition, we allow $\emptyset$ as a regex (with $\mathsf{var}(\emptyset) = \emptyset$), but we do not allow $\emptyset$ to occur in any other regex. An $\alpha \in \mathsf{RX}$ with $\mathsf{var}(\alpha) = \emptyset$ is called a *proper regular expression*, or just *regular expression*. We use REG to denote the set of all regular expressions.

We add and omit parentheses freely, as long as the meaning remains clear. We use the Kleene star α^* as shorthand for $\varepsilon \vee \alpha^+$, and A as shorthand for $\bigvee_{a \in A} a$ for non-empty $A \subseteq \Sigma$.

We define the semantics of regex using the *ref-words* (short for *reference words*) by Schmid [45]. A ref-word is a word over $(\Sigma \cup \Xi \cup \Gamma)$, where $\Gamma := \{[_x,]_x, \mid x \in \Xi\}$. Intuitively, the symbols $[_x$ and $]_x$ mark the beginning and the end of the match that is stored in the variable x, while an occurrence of x represents a reference to that variable. Instead of defining the language of a regex α directly, we first treat α as a generator of ref-words by defining its *ref-language* $\mathcal{R}(\alpha)$. If $\alpha \in \Sigma \cup \{\varepsilon\}$, $\mathcal{R}(\alpha) := \{\alpha\}$; and $\mathcal{R}(\&x) := \{x\}$ for all $x \in \Xi$. Furthermore, $\mathcal{R}(\alpha \cdot \beta) := \mathcal{R}(\alpha) \cdot \mathcal{R}(\beta)$, $\mathcal{R}(\alpha \vee \beta) := \mathcal{R}(\alpha) \cup \mathcal{R}(\beta)$, and $\mathcal{R}(\alpha^+) := \mathcal{R}(\alpha)^+$. Finally, $\mathcal{R}(\langle x \colon \alpha \rangle) := ([_x \mathcal{R}(\alpha)]_x)$. For regular expressions, $\mathcal{L}(\alpha) = \mathcal{R}(\alpha)$. Alternatively, $\mathcal{R}(\alpha) := \mathcal{L}(\alpha_{\mathcal{R}})$, where the proper regular expression $\alpha_{\mathcal{R}}$ is obtained by replacing each sub-regex $\langle x \colon \beta \rangle$ of α with $[_x \beta_{\mathcal{R}}]_x$, and each $\&x$ with x.

Intuitively speaking, every occurrence of a variable x in some $r \in \mathcal{R}(\alpha)$ functions as a pointer to the next factor $[_x v]_x$ to the left of this occurrence (or to ε if no such factor exists). In this way, a ref-word r compresses a word over Σ, the so-called dereference $\mathcal{D}(r)$ of r, which can be obtained by replacing every variable occurrence x by the corresponding factor v (note that v might again contain variable occurrences, which need to be replaced as well), and removing all symbols $[_x,]_x \in \Gamma$ afterwards. See [45] for a more detailed definition, or the following Example 2 for an illustration. Finally, we define $\mathcal{L}(\alpha) := \{\mathcal{D}(r) \mid r \in \mathcal{R}(\alpha)\}$.

▶ **Example 2.** Let $\alpha := \big(\langle x \colon (\mathtt{a} \vee \mathtt{b})^+ \rangle \&x\big)^+$. Then $\mathcal{R}(\alpha) = \{[_x w_1]_x \cdot x \cdots [_x w_n]_x \cdot x \mid n \geq 1, w_i \in \{\mathtt{a}, \mathtt{b}\}^+\}$. Hence, $\mathcal{L}(\alpha) = (L_{\mathsf{copy}})^+$, with $L_{\mathsf{copy}} := \{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^+\}$. Let

$\alpha_{\mathsf{sq}} := \big(\langle x\colon \& y\rangle\langle y\colon \& x\cdot \mathtt{a}\rangle\big)^*$. Then $\mathcal{R}(\alpha_{\mathsf{sq}}) = \{\big([_x y]_x \cdot [_y x \cdot \mathtt{a}]_y\big)^i \mid i \geq 0\}$. For example, consider the ref word $r_3 = [_x y]_x \cdot [_y x \cdot \mathtt{a}]_y \cdot [_x y]_x \cdot [_y x \cdot \mathtt{a}]_y \cdot [_x y]_x \cdot [_y x \cdot \mathtt{a}]_y$ with $\mathcal{D}(r_3) = \mathtt{a}^9$. Using induction, we can verify that $\mathcal{D}(r_i) = \mathtt{a}^{i^2}$. Thus, $\mathcal{L}(\alpha_{\mathsf{sq}}) = \{\mathtt{a}^{n^2} \mid n \geq 0\}$.

Hence, unlike regular expressions, regex can define non-regular languages. The expressive power comes at a price: their membership problem is NP-complete (follows from Angluin [3]), and various other problems are undecidable (Freydenberger [23]). Starting with Aho [2], there have been various approaches to specifying syntax and semantics of regex. While [2] only sketched the intuition behind the semantics, the first formal definition (using parse trees) was proposed by Câmpeanu, Salomaa, Yu [10], followed by the ref-words of Schmid [45]. For a comparison between these approaches and actual implementations, see the full version [26].

3 Memory Automata with Trap State

Memory automata [45] are a simple automaton model that characterizes $\mathcal{L}(\mathsf{RX})$. Intuitively speaking, these are classical finite automata that can record consumed factors in memories, which can be recalled later on in order to consume the same factor again. However, for our applications, we need to slightly adapt this model to *memory automata with trap-state*.

▶ **Definition 3.** For every $k \in \mathbb{N}$, a *k-memory automaton with trap-state*, denoted by $\mathsf{TMFA}(k)$, is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states* that contains the *trap-state* $[\mathsf{trap}]$, Σ is a finite *alphabet*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states* and $\delta\colon Q \times (\Sigma \cup \{\varepsilon\} \cup \{1, 2, \ldots, k\}) \to \mathcal{P}(Q \times \{\mathtt{o}, \mathtt{c}, \mathtt{r}, \diamond\}^k)$ is the *transition function* (where $\mathcal{P}(A)$ denotes the power set of a set A), which satisfies $\delta([\mathsf{trap}], b) = \{([\mathsf{trap}], \diamond, \diamond, \ldots, \diamond)\}$, for every $b \in \Sigma \cup \{\varepsilon\}$, and $\delta([\mathsf{trap}], i) = \emptyset$, for every i, $1 \leq i \leq k$. The elements $\mathtt{o}$, $\mathtt{c}$, $\mathtt{r}$ and $\diamond$ are called *memory instructions* (they stand for opening, closing and reseting a memory, respectively, and $\diamond$ leaves the memory unchanged).

A *configuration* of M is a tuple $(q, w, (u_1, r_1), \ldots, (u_k, r_k))$, where $q \in Q$ is the *current state*, w is the *remaining input* and, for every i, $1 \leq i \leq k$, (u_i, r_i) is the *configuration of memory* i, where $u_i \in \Sigma^*$ is the *content of memory* i and $r_i \in \{\mathtt{O}, \mathtt{C}\}$ is the *status of memory* i (i. e., $r_i = \mathtt{O}$ means that memory i is open and $r_i = \mathtt{C}$ means that it is closed). The *initial configuration* of M (*on input* w) is the configuration $(q_0, w, (\varepsilon, \mathtt{C}), \ldots, (\varepsilon, \mathtt{C}))$, a configuration $(q, w, (u_1, r_1), \ldots, (u_k, r_k))$ is an *accepting configuration* if $w = \varepsilon$ and $q \in F$.

M can change from a configuration $c = (q, vw, (u_1, r_1), \ldots, (u_k, r_k))$ to a configuration $c' = (p, w, (u'_1, r'_1), \ldots, (u'_k, r'_k))$, denoted by $c \vdash_M c'$, if there exists a transition $\delta(q, b) \ni (p, s_1, \ldots, s_k)$ with either ($b \in (\Sigma \cup \{\varepsilon\})$ and $v = b$) or ($b \in \{1, 2, \ldots, k\}$, $s_b = \mathtt{c}$ and $v = u_b$), and, for every i, $1 \leq i \leq k$,

$$\begin{array}{lll@{\qquad}lll} s_i = \diamond \wedge r_i = \mathtt{O} & \Rightarrow & (u'_i, r'_i) = (u_i v, r_i), & s_i = \diamond \wedge r_i = \mathtt{C} & \Rightarrow & (u'_i, r'_i) = (u_i, r_i), \\ s_i = \mathtt{o} & \Rightarrow & (u'_i, r'_i) = (v, \mathtt{O}), & s_i = \mathtt{c} & \Rightarrow & (u'_i, r'_i) = (u_i, \mathtt{C}), \\ s_i = \mathtt{r} & \Rightarrow & (u'_i, r'_i) = (\varepsilon, \mathtt{C}). & & & \end{array}$$

Furthermore, M can change from a configuration $(q, vw, (u_1, r_1), \ldots, (u_k, r_k))$ to the configuration $([\mathsf{trap}], w, (u_1, r_1), \ldots, (u_k, r_k))$, if $\delta(q, b) \ni (p, s_1, \ldots, s_k)$ for some $p \in Q$, $b \in \{1, 2, \ldots, k\}$ and $s_b = \mathtt{c}$, such that $u_b = vv'$ with $v' \neq \varepsilon$ and $v'[1] \neq w[1]$.

A transition $\delta(q, b) \ni (p, s_1, s_2, \ldots, s_k)$ is an *ε-transition* if $b = \varepsilon$ and is called *consuming*, otherwise (if all transitions are consuming, then M is called *ε-free*). If $b \in \{1, 2, \ldots, k\}$, it is called a *memory recall transition* and the situation that a memory recall transition leads to the state $[\mathsf{trap}]$, is called a *memory recall failure*.

The symbol $\vdash^*_M$ denotes the reflexive and transitive closure of $\vdash_M$. A $w \in \Sigma^*$ is *accepted* by M if $c_{\text{init}} \vdash^*_M c_f$, where c_{init} is the initial configuration of M on w and c_f is an accepting configuration. The set of words accepted by M is denoted by $\mathcal{L}(M)$.

Note that executing the open action $\mathtt{o}$ on a memory that already contains some word discards the previous contents of that memory. For illustrations and examples for TMFA, we refer to [45]. A crucial part of TMFA is the trap-state [trap], in which computations terminate, if a memory recall failure happens. If [trap] is not accepting, then TMFA are (apart from negligible formal differences) identical to the memory automata introduced in [45], which characterize the class of regex language. If, on the other hand, [trap] is accepting, then every computation with a memory recall failure is accepting (independent from the remaining input). While it seems counter-intuitive to define the words of a language via “failed” back-references, the possibility of having an accepting trap-state yields closure under complement for *deterministic* TMFA (see Theorem 6). It will be convenient to consider the partition of TMFA into $\mathsf{TMFA}^{\mathsf{rej}}$ and $\mathsf{TMFA}^{\mathsf{acc}}$ (having a rejecting and an accepting trap-state, respectively).

Every $\mathsf{TMFA}^{\mathsf{acc}}$ can be transformed into an equivalent $\mathsf{TMFA}^{\mathsf{rej}}$, which implies $\mathcal{L}(\mathsf{TMFA}) = \mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}})$; thus, it follows from [45] that TMFA characterize $\mathcal{L}(\mathsf{RX})$. The idea of this construction is as follows. Every memory i is simulated by two memories $(i, 1)$ and $(i, 2)$, which store a (nondeterministically guessed) factorisation of the content of memory i. This allows us to guess and verify if a memory recall failure occurs, i. e., $(i, 1)$ stores the longest prefix that can be matched and $(i, 2)$ starts with the first mismatch. For correctness, it is crucial that every possible factorisation of the content of a memory i can be guessed.

▶ **Theorem 4.** $\mathcal{L}(\mathsf{TMFA}) = \mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}}) = \mathcal{L}(\mathsf{RX})$.

A consequence of the proof is that TMFA inherits the NP-hardness of the membership problem from RX. We do not devote more attention to this, as we focus on deterministic TMFA: A TMFA is *deterministic* (or a DTMFA, for short) if δ satisfies $|\delta(q, b)| \leq 1$, for every $q \in Q$ and $b \in \Sigma \cup \{\varepsilon\} \cup \{1, 2, \ldots, k\}$ (for the sake of convenience, we then interpret δ as a partial function with range $Q \times \{\mathtt{o}, \mathtt{c}, \mathtt{r}, \diamond\}^k$), and, furthermore, for every $q \in Q$, if $\delta(q, x)$ is defined for some $x \in \{1, 2, \ldots, k\} \cup \{\varepsilon\}$, then, for every $y \in (\Sigma \cup \{\varepsilon\} \cup \{1, 2, \ldots, k\}) \setminus \{x\}$, $\delta(q, y)$ is undefined. Analogously to TMFA, we partition DTMFA into $\mathsf{DTMFA}^{\mathsf{acc}}$ and $\mathsf{DTMFA}^{\mathsf{rej}}$.

The algorithmically most important feature of DTMFA is that their membership can be solved efficiently by running the automaton on the input word. However, for each processed input symbol, there might be a delay of at most $|Q|$ steps, due to ε-transitions and recalls of empty memories, which leads to $O(|Q||w|)$. Removing such non-consuming transitions first, is possible, but problematic. In particular, recalls of empty memories depend on the specific input word and could only be determined beforehand by storing for each memory whether it is empty, which is too expensive. However, by $O(|Q|^2)$ preprocessing, we can compute the information that is needed in order to determine in $O(k)$ where to jump if certain memories are empty, and which memories are currently empty can be determined on-the-fly while processing the input. This leads to a delay of only k, the number of memories:

▶ **Theorem 5.** *Given $M \in \mathsf{DTMFA}$ with n states and k memories, and $w \in \Sigma^*$, we can decide in time $O(n^2 + k|w|)$, whether or not $w \in \mathcal{L}(M)$.*

Note that the preprocessing in the proof of Theorem 5 is only required once, so we can solve the membership for several words w_i in $O(n^2 + k \sum |w_i|)$. Moreover, if it is guaranteed that no empty memories are recalled, then membership can be solved in $O(n + |w|)$ (where $O(n)$ is needed in order to remove ε-transitions).

Similar to DFA, it is possible to complement DTMFA by toggling the acceptance of states. However, for DTMFA, we have to remove ε-transitions and recalls of empty memories. In particular, the construction for Theorem 6 uses the finite control to store whether memories are empty or not, which causes a blow-up that is exponential in the number of memories.

▶ **Theorem 6.** $\mathcal{L}(\mathsf{DTMFA})$ *is closed under complement.*

We next discuss expressive power: If there is a constant upper bound on the lengths of contents of memories that are recalled in accepting computations of an $M \in \mathsf{DTMFA}$, then memories can be simulated by the finite state control; thus, $\mathcal{L}(M) \in \mathcal{L}(\mathsf{REG})$. Consequently, if $\mathcal{L}(M) \notin \mathcal{L}(\mathsf{REG})$, there is a word uvw that is accepted by recalling some memory with an arbitrarily large content v. Moreover, if $[\mathsf{trap}]$ is non-accepting, then no word can be accepted that contains u as a prefix, but not uv, since this will cause a memory recall failure. Intuitively speaking, a $\mathsf{DTMFA}^{\mathsf{rej}}$ for a non-regular language makes arbitrarily large "jumps":

▶ **Lemma 7** (Jumping Lemma). *Let $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$. Then either L is regular, or for every $m \geq 0$, there exist $n \geq m$ and $p_n, v_n \in \Sigma^+$ such that **1.** $|v_n| = n$, **2.** v_n is a factor of p_n, **3.** $p_n v_n$ is a prefix of a word from L, **4.** for all $u \in \Sigma^+$, $p_n u \in L$ only if v_n is a prefix of u.*

▶ **Example 8.** Let $L := \{ww \mid w \in \Sigma^*\}$ with $|\Sigma| \geq 2$, which is well-known to be not regular. Assume $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ and choose $m := 1$. Then there exist $n \geq 1$ and $p_n, v_n \in \Sigma^*$ that satisfy the conditions of Lemma 7. Choose $a \in \Sigma$ that is not the first letter of v_n, and define $u := a p_n a$. Then v_n is not a prefix of u, but $p_n u = (p_n a)^2 \in L$, which is a contradiction.

▶ **Example 9.** Let $L := \{\mathtt{a}^i \mathtt{b} \mathtt{a}^j \mid i > j \geq 0\}$. Using textbook methods, it is easily shown that L is not regular. Now, assuming that $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$, choose $m := 4$. Then there exist $n \geq 4$ and $p_n, v_n \in \Sigma^+$ that satisfy the conditions of Lemma 7. As $p_n v_n$ is a prefix of a word in L, either $p_n = \mathtt{a}^i$ or $p_n = \mathtt{a}^i \mathtt{b} \mathtt{a}^j$ with $i, j \geq 0$ (and $i \geq 4$ or $i + j \geq 3$). In the first case, consider $u := \mathtt{ba}$. Then $p_n u = \mathtt{a}^i \mathtt{ba}$ with $i \geq 4$; hence, $p_n u \in L$. But u starts with $\mathtt{b}$, and v_n is a factor of $p_n = \mathtt{a}^i$. Contradiction, as v_n cannot be a prefix of u. For the second case, let $u := \mathtt{a}$. As $p_n v_n$ is a prefix of a word in L, and as $|v_n| = n$, $i > j + n \geq j + 4$ must hold. Hence, $p_n u = \mathtt{a}^i \mathtt{b} \mathtt{a}^{j+1}$, and $p_n u \in L$. Contradiction, as v_n is not a prefix of u.

For unary languages, there is an alternative to Lemma 7 that is easier to apply and characterizes unary $\mathsf{DTMFA}^{\mathsf{rej}}$-languages. It is built on the following definition: A language $L \subseteq \{\mathtt{a}\}^*$ is an *infinite arithmetic progression* if $L = \{\mathtt{a}^{bi+c} \mid i \geq 0\}$ for some $b \geq 1$, $c \geq 0$.

▶ **Lemma 10.** *Let $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ be an infinite language with $L \subseteq \{\mathtt{a}\}^*$. The following conditions are equivalent: **1.** L is regular. **2.** L contains an infinite arithmetic progression. **3.** There is $b \geq 1$ such that, for every $n \geq 0$, $\mathtt{a}^{bi+c_n} \in L$ for some $c_n \geq 0$ and all $0 \leq i \leq n$.*

▶ **Example 11.** Let $\alpha := \langle x \colon \mathtt{aa}^+ \rangle (\& x)^+$ (this regex is also known as "Abigail's expression" [1] in the PERL community). Then $\mathcal{L}(\alpha) = \{\mathtt{a}^{mn} \mid m, n \geq 2\}$. In other words, α generates the language of all $\mathtt{a}^i$ such that i is a composite number (i. e., not a prime number). As $\mathcal{L}(\alpha)$ is not regular and contains the arithmetic progression $2i+4$, Lemma 10 yields $\mathcal{L}(\alpha) \notin \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$.

The following result is a curious consequence of Lemma 10:

▶ **Proposition 12.** *Over unary alphabets, $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \cap \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}}) = \mathcal{L}(\mathsf{REG})$.*

4 Deterministic Regex

In order to define deterministic regex as an extension of deterministic regular expressions, we first extend the notion of a *marked alphabet* that is commonly used for the latter: For every alphabet A, let $\tilde{A} := \{a_{(n)} \mid a \in A, n \geq 1\}$. For every $\alpha \in \mathsf{RX}$, we define $\tilde{\alpha}$ as a regex that is obtained by taking $\alpha_{\mathcal{R}}$ (the proper regular expression over $\Sigma \cup \Xi \cup \Gamma$ that generates the ref-language $\mathcal{R}(\alpha)$), and marking each occurrence of $\chi \in (\Sigma \cup \Xi \cup \Gamma)$

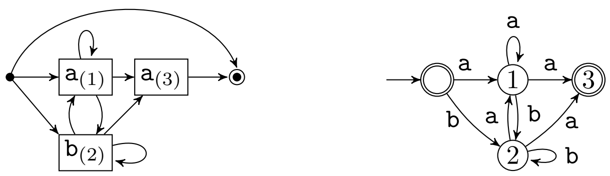
by a unique number (to make this well-defined, we assume that the markings start at 1 and are increased stepwise). For example, if $\alpha := \langle y \colon (\mathtt{a} \vee \&x)^* \cdot (\varepsilon \vee \mathtt{b} \cdot \mathtt{a})\rangle \cdot \&y$, then $\tilde{\alpha} = [_{y\,(1)}(\mathtt{a}_{(2)} \vee x_{(3)})^* \cdot (\varepsilon \vee \mathtt{b}_{(4)} \cdot \mathtt{a}_{(5)})]_{y\,(6)} \cdot y_{(7)}$. We also use these markings in the ref-words: For example, $[_{y\,(1)}\mathtt{a}_{(2)}\mathtt{a}_{(2)}x_{(3)}\mathtt{a}_{(2)}]_{y\,(6)}y_{(7)} \in \mathcal{R}(\tilde{\alpha})$.

Before we explain this definition and use it to define deterministic regex, we first discuss the special case of deterministic regular expressions: A proper regular expression α is *not* deterministic if there exist words $u, v_1, v_2 \in \tilde{\Sigma}^*$, a terminal $a \in \Sigma$ and positions $i \neq j$ such that $ua_{(i)}v_1$ and $ua_{(j)}v_2$ are elements of $\mathcal{L}(\tilde{\alpha})$ (see e. g. [9, 31]). Otherwise, it is a *deterministic proper regular expression* (or, for short, just *deterministic regular expression*).

The intuition behind this definition is based on the Glushkov construction for the conversion of regular expressions into finite automata, as a regular expression α is deterministic if and only if its *Glushkov automaton* $\mathcal{M}(\alpha)$ is deterministic. Given a regular expression α, we define $\mathcal{M}(\alpha)$ in the following way: First, we use the marked regular expression $\tilde{\alpha}$ to construct its *occurrence graph*[1] $G_{\tilde{\alpha}}$, a directed graph that has a source node src, a sink node snk, and one node for each $a_{(i)}$ in $\tilde{\alpha}$. The edges are constructed in the following way: Each node $a_{(i)}$ has an incoming edge from src if $a_{(i)}$ can be the first letter of a word in $\mathcal{L}(\tilde{\alpha})$, and an outgoing edge to snk if it can be the last letter of such a word. Furthermore, for each factor $a_{(i)}b_{(j)}$ that occurs in a word of $\mathcal{L}(\tilde{\alpha})$, there is an edge from $a_{(i)}$ to $b_{(j)}$. As a consequence, there is a one-to-one-correspondence between marked words in $\mathcal{L}(\tilde{\alpha})$ and paths from src to snk in $G_{\tilde{\alpha}}$. To obtain $\mathcal{M}(\alpha)$, we directly interpret $G_{\tilde{\alpha}}$ as NFA over Σ: The source src is the starting state, each node $a_{(i)}$ is a state q_i, and an edge from $a_{(i)}$ to $b_{(j)}$ corresponds to a transition from q_i to q_j when reading b. The sink snk does not become a state; instead, each node with an edge to snk is a final state (hence, $\mathcal{M}(\alpha)$ contains the source state, and one state for every terminal in α). This interpretation allows us to treat occurrence graphs as an alternative notation for a subclass of NFA (namely those where the starting state is not reachable from other states, and for each state q, there is a characteristic terminal a_q such that all transitions to q read a_q). When doing so, we usually omit the occurrence markings on the nodes in graphical representations.

Intuitively, $\mathcal{M}(\alpha)$ treats each terminal of α as a state. Recall that α is not deterministic if there exists words $ua_{(i)}v_1$ and $ua_{(j)}v_2$ in $\mathcal{L}(\tilde{\alpha})$ with $i \neq j$. This corresponds to the situation where, after reading u, $\mathcal{M}(\alpha)$ has to decide between states $a_{(i)}$ and $a_{(j)}$ for the input letter a.

▶ **Example 13.** Let $\alpha := (\varepsilon \vee ((\mathtt{a} \vee \mathtt{b})^+ \mathtt{a}))$. Then $\tilde{\alpha} = (\varepsilon \vee ((\mathtt{a}_1 \vee \mathtt{b}_2)^+ \mathtt{a}_3))$, and $\mathcal{M}(\alpha)$, the Glushkov automaton of α, is defined as follows:



To the left, $\mathcal{M}(\alpha)$ is represented as an occurrence graph, to the right in standard NFA notation. Then $\mathcal{M}(\alpha)$ and α are both not deterministic: For $\mathcal{M}(\alpha)$, consider state 1; for α, consider $u = \mathtt{a}_{(1)}$, $v_1 = \mathtt{a}_{(3)}$, $v_2 = \varepsilon$, and the words $u\mathtt{a}_{(1)}v_1$ and $u\mathtt{a}_{(3)}v_2$.

[1] Most literature, like [9], defines the occurrence graph only implicitly by using sets first, last, and follow, which correspond to the edge from src, the edges to snk, or to the other edges of the graph, respectively. The explicit use of a graph is taken from the k-occurrence automata by Bex et al. [5]. We shall see that an advantage of graphs is that they can be easily extended by describing memory actions to the edges.

As shown in [9], $\mathcal{L}(\mathsf{DREG}) \subset \mathcal{L}(\mathsf{REG})$ (also see [16, 39], or Lemma 23 below). Like for determinism of regular expressions, the key idea behind our definition of deterministic regex is that a matcher for the expression treats terminals (and variable references) as states. Then an expression is deterministic if the current symbol of the input word always uniquely determines the next state and all necessary variable actions. For regular expressions, non-determinism can only occur when the matcher has to decide between two occurrences of the same terminal symbol; but as regex also need to account for non-determinism that is caused by variable operations or references, their definition of non-determinism is more complicated.

▶ **Definition 14.** An $\alpha \in \mathsf{RX}$ is *not deterministic* if there exist $\rho_1, \rho_2 \in \mathcal{R}(\tilde{\alpha})$ such that any of the following conditions is met for some $r, s_1, s_2 \in (\tilde{\Sigma} \cup \tilde{\Xi} \cup \tilde{\Gamma})^*$ and $\gamma_1, \gamma_2 \in \tilde{\Gamma}^*$:
1. $\rho_1 = r \cdot \gamma_1 \cdot a_{(i)} \cdot s_1$ and $\rho_2 = r \cdot \gamma_2 \cdot a_{(j)} \cdot s_2$ with $a \in \Sigma$ and $i \neq j$,
2. $\rho_1 = r \cdot \gamma_1 \cdot x_{(i)} \cdot s_1$ and $\rho_2 = r \cdot \gamma_2 \cdot \chi_{(j)} \cdot s_2$ with $x \in \Xi$, $\chi \in (\Sigma \cup \Xi)$ and $i \neq j$,
3. $\rho_1 = r \cdot \gamma_1 \cdot \chi_{(i)} \cdot s_1$ and $\rho_2 = r \cdot \gamma_2 \cdot \chi_{(i)} \cdot s_2$ with $\chi \in (\Sigma \cup \Xi)$ and $\gamma_1 \neq \gamma_2$,
4. $\rho_1 = r \cdot \gamma_1$ and $\rho_2 = r \cdot \gamma_2$ with $\gamma_1 \neq \gamma_2$.

Otherwise, α is *deterministic*. We use DRX to denote the set of all deterministic regex, and define $\mathsf{DREG} := \mathsf{DRX} \cap \mathsf{REG}$ as the set of deterministic regular expressions.

▶ **Example 15.** Let $\alpha_1 := (\langle x\colon \mathtt{a}\rangle \vee \mathtt{a})$, $\alpha_2 := (\mathtt{a} \vee \&x)$, $\alpha_3 := (\langle x\colon \varepsilon\rangle \vee \varepsilon)\mathtt{a}$, $\alpha_4 := (\langle x\colon \varepsilon\rangle \vee \varepsilon)$. None of these regex are deterministic, as each α_i meets the i-th condition of Definition 14. We discuss this for α_1: Observe $\tilde{\alpha}_1 = ([_{x(1)}\mathtt{a}_{(2)}]_{x(3)}) \vee \mathtt{a}_{(4)}$. Then choosing $\rho_1 = [_{x(1)}\mathtt{a}_{(2)}]_{x(3)}$ and $\rho_2 = \mathtt{a}_{(4)}$, with $r = \varepsilon$, $\gamma_1 = [_{x(1)}$, $s_1 =]_{x(3)}$, and $\gamma_2 = s_2 = \varepsilon$ shows the condition is met.

Let $\beta_1 := \langle x\colon (\mathtt{a} \vee \mathtt{b})^*\rangle \mathtt{c} \cdot \&x$ and $\beta_2 := \big(\langle x\colon \&y\rangle \langle y\colon \&x \cdot \mathtt{a}\rangle\big)^*$. Both regex are deterministic, with $\mathcal{L}(\beta_1) := \{w\mathtt{c}w \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$ and $\mathcal{L}(\beta_2) = \{\mathtt{a}^{n^2} \mid n \geq 0\}$ (see Example 2).

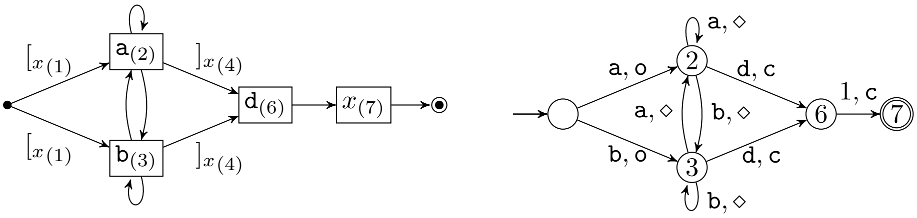
Condition 1 of Definition 14 describes cases where non-determinism is caused by two occurrences of the same terminal (γ_1 and γ_2 are included for cases like α_1 in Example 15). If restricted to regular expressions, it is equivalent to the usual definition of deterministic regular expressions. Condition 2 expresses that the matcher has to decide between a variable reference and any other symbol; while in condition 3, the symbol is unique, but there is a non-deterministic choice between variable operations. Finally, condition 4 describes cases where the behavior of variables is non-deterministic after the end of the word (while one could consider this edge case deterministic, this choice simplifies recursive definitions). In conditions 3 and 4, the definition not only requires that it is clear which variables are reset, but also that it is clear which part of the regex acts on the variables. Hence, $(\langle x\colon \varepsilon\rangle \vee \langle x\colon \varepsilon\rangle)$ is also not deterministic. This is similar to the notion of strong determinism for regular expressions, see [29]. As one might expect, some non-deterministic regexes define DRX-languages:

▶ **Example 16.** Let $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ and $\alpha := \mathtt{1}^+\langle x\colon \mathtt{0}^*\rangle(\mathtt{1}^+\&x)^*\mathtt{1}^+$. This regex was introduced by Fagin et al. [20], who call its language the "uniform-0-chunk language". Obviously, α is not deterministic (in fact, it satisfies conditions 1, 2, and 3 of Definition 14). Nonetheless, it is possible to express $\mathcal{L}(\alpha)$ with the deterministic regex $\mathtt{1}\big(\mathtt{1}^+ \vee \big(0\langle x\colon \mathtt{0}^*\rangle 1^+(0 \cdot \&x \cdot 1^+)^*\big)\big)$.

We now discuss the conversion from DRX to $\mathsf{DTMFA}^{\mathsf{rej}}$, which generalizes the Glushkov construction of $\mathcal{M}(\alpha)$ for regular expressions. The core idea is extending the occurrence graph to a *memory occurrence graph* $G_{\tilde{\alpha}}$, which has two crucial differences: First, instead of only considering terminals, each terminal and each variable reference of a regex α becomes a node. Second, each edge is labeled with a ref-word from $\tilde{\Gamma}^*$ that describes the memory actions (hence, there can be multiple edges from one node to another). In analogy to the occurrence graph, each memory occurrence graph can be directly interpreted as an ε-free $\mathsf{TMFA}^{\mathsf{rej}}$.

▶ **Theorem 17.** *Let $\alpha \in$ RX, and let n denote the number of occurrences of terminals and variable references in α. We can construct an $n{+}2$ state* TMFA$^{\mathsf{rej}}$ $\mathcal{M}(\alpha)$ *with* $\mathcal{L}(\mathcal{M}(\alpha)) = \mathcal{L}(\alpha)$ *that is deterministic if and only if α is deterministic. In time $O(|\Sigma||\alpha|n)$, the algorithm either **1.** computes $\mathcal{M}(\alpha)$ if α is deterministic, or **2.** detects that α is not deterministic.*

▶ **Example 18.** Consider the deterministic regex $\alpha := \langle x\colon (\mathtt{a} \vee \mathtt{b})^{+}\rangle \cdot \mathtt{d} \cdot \& x$. Applying the markings yields $\tilde{\alpha} := [_{x\,(1)}(\mathtt{a}_{(2)} \vee \mathtt{b}_{(3)})^{+}]_{x\,(4)} \cdot \mathtt{d}_{(6)} \cdot x_{(7)}$, and $\mathcal{M}(\alpha)$ is the following automaton:



To the left, $\mathcal{M}(\alpha)$ is represented as the memory occurrence graph $G_{\tilde{\alpha}}$, to the right as the DTMFA that can be directly derived from this graph (which uses memory 1 for x).

The construction from the proof of Theorem 17 behaves like the Glushkov construction for regular expressions, with one important difference: On regex that are not deterministic, its running time may be exponential in the number of variables; as there are non-deterministic regex where conversion into a TMFA without ε-transitions requires an exponential amount of transitions. E. g., for $k \geq 1$, let $\alpha := \mathtt{a} \cdot (\varepsilon \vee \langle x_1\colon \varepsilon\rangle) \cdots (\varepsilon \vee \langle x_k\colon \varepsilon\rangle) \cdot \mathtt{b}$ and $\beta := \mathtt{a}\big(\bigvee_{1 \leq i \leq k} \langle x_i\colon \varepsilon\rangle\big)^{*}\mathtt{b}$. An automaton that is derived with a Glushkov style conversion then contains states q_1 and q_2 that correspond to the terminals; and between these two states, there must be 2^k different transitions to account for all possible combinations of actions on the variables. This suggests that converting a regex into a TMFA without ε-edges is only efficient for deterministic regex; while in general, it is probably advisable to use a construction with ε-edges.

By combining Theorems 17 and 5, due to $n \leq |\alpha|$, we immediately obtain the following:

▶ **Theorem 19.** *Given $\alpha \in$ DRX with n occurrences of terminal symbols or variable references and k variables, and $w \in \Sigma^*$, we can decide in time $O(|\Sigma||\alpha|n + k|w|)$, whether $w \in \mathcal{L}(\alpha)$.*

If we ensure that recalled variables never contain ε (or that only a bounded number of variables references are possible in a row), we can even drop the factor k. For comparison, the membership problem for DREG can be decided in time $O(|\Sigma||\alpha| + |w|)$ when using optimized versions of the Glushkov construction (see [8, 43]), and in $O(|\alpha| + |w| \cdot \log\log|\alpha|)$ with the algorithm by Groz, Maneth, and Staworko [31] that does not compute an automaton.

5 Expressive Power

While Câmpeanu, Salomaa, Yu [10] and Carle and Narendran [11] state pumping lemmas for a class of regex, these do not apply to regex as defined in this paper. However, Lemmas 7 and 10, introduced in Section 3, shall be helpful for proving inexpressibility. A consequence of Lemma 10 is that there are infinite unary DTMFA$^{\mathsf{rej}}$-languages that are not pumpable (in the sense that certain factors can be repeated arbitrarily often), as this would always lead to an arithmetic progression. It is also possible to demonstrate this phenomenon on larger alphabets, without relying on a trivial modification of the unary case:

▶ **Example 20.** The Fibonacci word F_ω is the infinite word that is the limit of the sequence of words $F_0 := \mathtt{b}$, $F_1 := \mathtt{a}$, and $F_{n+2} := F_{n+1} \cdot F_n$ for all $n \geq 0$. The Fibonacci word has a number of curious properties. In particular, it includes no cubes (i. e., factors www, with $w \neq \varepsilon$). This and various other properties are explained throughout Lothaire [38]. Let

$$\alpha := \mathtt{a}\langle x_0 : \mathtt{b}\rangle\langle x_1 : \mathtt{a}\rangle\big(\langle x_2 : \&x_1\&x_0\rangle\langle x_3 : \&x_1\&x_0\&x_1\rangle\langle x_0 : \&x_3\&x_2\rangle\langle x_1 : \&x_3\&x_2\&x_3\rangle\big)^*.$$

Then $\mathcal{L}(\alpha) = \{F_{4i+3} \mid i \geq 0\}$. Hence, the words of $\mathcal{L}(\alpha)$ converge towards F_ω. The proof of this equivalence is straightforward, but long. It uses that $F_{n+3} = F_{n+1} \cdot F_n \cdot F_{n+1}$ holds for all $n \geq 0$ As F_ω contains no cube, the same applies to all F_n. Thus, $\mathcal{L}(\alpha)$ is a DRX-language that cannot be pumped by repeating factors of sufficiently large words arbitrarily often.

For further separations, we use the following language:

▶ **Example 21.** Let $\alpha := \mathtt{a}^2 \cdot \langle x : \mathtt{a}^2\rangle \cdot \big(\langle y : \&x \cdot \&x\rangle \cdot \langle x : \&y \cdot \&y\rangle\big)^*$. Then $\mathcal{L}(\alpha) = \{\mathtt{a}^{4^i} \mid i \geq 1\}$.

From this, we define an $L \in \mathcal{L}(\mathsf{TMFA})$ with neither $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$, nor $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$:

▶ **Lemma 22.** *Let* $L := \{\mathtt{a}^{4i+1} \mid i \geq 0\} \cup \{\mathtt{a}^{4^i} \mid i \geq 1\}$. *Then* $L \in \mathcal{L}(\mathsf{TMFA}) \setminus \mathcal{L}(\mathsf{DTMFA})$.

While $\mathsf{DTMFA}^{\mathsf{rej}}$-inexpressibility provides us with a powerful sufficient criterion for DRX-inexpressibility, it is not powerful enough to cover all cases of DRX-inexpressibility. In particular, there are even regular languages that are no DRX-languages:

▶ **Lemma 23.** *Let* $L := \mathcal{L}\big((\mathtt{ab})^*(\mathtt{a} \vee \varepsilon)\big) = \{(\mathtt{ab})^{\frac{1}{2}i} \mid i \geq 0\}$. *Then* $L \in \mathcal{L}(\mathsf{REG}) \setminus \mathcal{L}(\mathsf{DRX})$.

The language L from Lemma 23 is also known to be a non-deterministic regular language (see e. g. [9]). Our proof can be seen as taking the idea behind the characterization of deterministic regular languages from [9], applying it to the specific language L, and also taking variables into account. While this accomplishes the task of proving that deterministic regex share some of the limitations of deterministic regular expressions, the approach does not generalize (at least not in a straightforward manner). In particular, deterministic regex can express regular languages that are not deterministic regular, and are also quite similar to L:

▶ **Example 24.** Let $L := \{(\mathtt{ab})^{\frac{3}{2}i} \mid i \geq 0\}$. Then L is generated by the non-deterministic regular expression $(\mathtt{ababab})^*(\varepsilon \vee (\mathtt{aba}))$, and one can show that L is not a deterministic regular language by using the BKW-algorithm [9] (also [16, 39]) on the minimal DFA M for L. But for $\alpha := \mathtt{a}\langle y : \mathtt{b}\rangle\langle x : \mathtt{a}\rangle\big(\langle z : \&y\rangle\langle y : \&x\rangle\langle x : \&z\rangle\big)^*$, $\alpha \in \mathsf{DRX}$ and $\mathcal{L}(\alpha) = L$.

The "shifting gadget" that is used in Example 24 can be extended to show a far more general result for unary languages. Considering that $\mathcal{L}(\mathsf{DREG}) \subset \mathcal{L}(\mathsf{REG})$ holds even over unary alphabets (cf. Losemann et al. [37]), the following result might seem surprising:

▶ **Theorem 25.** *For every regular language* L *over a unary alphabet,* $L \in \mathcal{L}(\mathsf{DRX})$.

As a DFA with n states is converted into a deterministic regex of length $O(n)$, this construction is even efficient. We summarize our observations (also see Figure 1):

▶ **Theorem 26.** $\mathcal{L}(\mathsf{DREG}) \subset \mathcal{L}(\mathsf{DRX}) \subset \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \subset \mathcal{L}(\mathsf{DTMFA}) \subset \mathcal{L}(\mathsf{TMFA}) = \mathcal{L}(\mathsf{RX})$.
The following pairs of classes are incomparable: $\mathcal{L}(\mathsf{DRX})$ *and* $\mathcal{L}(\mathsf{REG})$, $\mathcal{L}(\mathsf{DRX})$ *and* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$, *as well as* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ *and* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$.

We can also use the examples from this section to show that $\mathcal{L}(\mathsf{DRX})$ and $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ are not closed under most of the commonly studied operations on languages:

▶ **Theorem 27.** $\mathcal{L}(\mathsf{DRX})$ *and* $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ *are not closed under the following operations: union, concatenation, reversal, complement, homomorphism, and inverse homomorphism.* $\mathcal{L}(\mathsf{DRX})$ *is also not closed under intersection, and intersection with* DREG*-languages.*

We leave open whether $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ is closed under intersection (with itself or with $\mathcal{L}(\mathsf{DREG})$), but we conjecture that this is not the case. We also leave open whether $\mathcal{L}(\mathsf{DRX})$ and $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ are closed under Kleene plus or star.

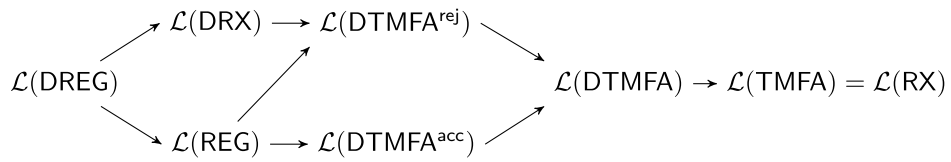


Figure 1 The proper inclusions from Theorem 26. Arrows point from sub- to superset.

6 Two Variants of Determinism

In this section, we examine a restriction and an extension of DRX and DTMFA. We begin with the restriction, which we motivate with the following observation: As shown by Carle and Narendran [11], the intersection problem for regex is undecidable. For DRX, that proof cannot be used, but the result still holds (and by Theorem 17, this extends to DTMFA):

▶ **Theorem 28.** *Given $\alpha, \beta \in$ DRX, it is undecidable whether $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta) = \emptyset$.*

As a consequence, DTMFA intersection emptiness problem is also undecidable. Theorem 28 applies even to very restricted DRX, as no variable binding contains a reference to another variable, $|\mathsf{var}(\alpha)| = 2$, and $|\mathsf{var}(\beta)| = 3$. Hence, bounding the number of variables does not make the problem decidable. Instead, the key part seems to be that the variables occur under Kleene stars, which means that they can be reassigned an unbounded amount of times. Following similar observations, Freydenberger and Holldack [25] introduced the following concept: A regex is *variable-star-free (vstar-free)* if each of its plussed sub-regexes contains neither variable references, nor variable bindings. Analogously, we call a TMFA *memory-cycle-free* if it contains no cycle with a *memory transition* (a transition in a TMFA that is a memory recall, or that contains memory actions other than $\diamond$). Let $\mathsf{RX}_{\mathsf{vsf}}$ be the set of all vstar-free regex, and $\mathsf{DRX}_{\mathsf{vsf}} = \mathsf{RX}_{\mathsf{vsf}} \cap \mathsf{DRX}$. Let $\mathsf{TMFA}_{\mathsf{mcf}}$ be the set of all memory-cycle-free TMFA, and define $\mathsf{DTMFA}_{\mathsf{mcf}}$, $\mathsf{TMFA}^{\mathsf{rej}}_{\mathsf{mcf}}, \ldots$ analogously. The proof of Theorem 17 allows us to conclude that $\mathcal{M}(\alpha) \in \mathsf{DTMFA}_{\mathsf{mcf}}$ holds for every $\alpha \in \mathsf{DRX}_{\mathsf{vsf}}$. Likewise, we can use the proof of Theorem 4 to conclude $\mathcal{L}(\mathsf{TMFA}_{\mathsf{mcf}}) = \mathcal{L}(\mathsf{RX}_{\mathsf{vsf}})$. Note that for ε-free $\mathsf{DTMFA}_{\mathsf{mcf}}$, the membership problem can be decided in time $O(|Q| + |w|)$, as the preprocessing step of Theorem 5 is not necessary (as only a bounded number of variable references is possible in each run). Likewise, we can drop the factor k from Theorem 19 when restricted to $\mathsf{DRX}_{\mathsf{vsf}}$.

As shown by Freydenberger [24], it is decidable in PSPACE whether $\bigcap_{i=1}^{n} \mathcal{L}(\alpha_i) = \emptyset$ for $\alpha_1, \ldots, \alpha_n \in \mathsf{RX}_{\mathsf{vsf}}$. By combining the proof for this with some ideas from another construction from [24], we encode the intersection emptiness problem for $\mathsf{TMFA}_{\mathsf{mcf}}$ in the *existential theory of concatenation with regular constraints* (a PSPACE-decidable, positive logic on words, see Diekert [17], Diekert, Jeż, Plandowski [19]). This yields the following:

▶ **Theorem 29.** *Given $M_1, \ldots, M_n \in \mathsf{TMFA}_{\mathsf{mcf}}$, we can decide whether $\bigcap_{i=1}^{n} \mathcal{L}(M_i) = \emptyset$ in* PSPACE*. The problem is* PSPACE*-hard, even if restricted to $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$, $\alpha \in \mathsf{DRX}_{\mathsf{vsf}}$ and $\beta \in$ DREG (if the size of Σ is not bounded), or to $\mathcal{L}(\alpha) \cap \mathcal{L}(M)$, $\alpha \in \mathsf{DRX}_{\mathsf{vsf}}$ and $M \in$ DFA.*

The unbounded size of Σ comes from the PSPACE-hardness of the intersection emptiness problem for DRX by Martens et al. [40], which has the same requirement. Using the existential theory of concatenation for the upper bound might seem conceptually excessive – but this cannot be avoided (see Section A.18 in the full version of the paper [26]).

We now combine the proofs of Theorems 6 and 29, and observe:

▶ **Theorem 30.** *Given $M_1, M_2 \in \mathsf{DTMFA}_{\mathsf{mcf}}$, $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$ can be decided in* PSPACE*.*

Obviously, this implies that equivalence for $\mathsf{DTMFA}_{\mathsf{mcf}}$ is decidable in PSPACE, and, furthermore, this also holds for $\mathsf{DRX}_{\mathsf{vsf}}$, which is an interesting contrast to non-deterministic $\mathsf{RX}_{\mathsf{vsf}}$: As shown by Freydenberger [23], equivalence (and, hence, inclusion and minimization) are undecidable for $\mathsf{RX}_{\mathsf{vsf}}$ (while [23] does not explicitly mention the concept, the regex in that proof are vstar-free, as discussed in [25]). Hence, Theorem 30 also yields a minimization algorithm for $\mathsf{DRX}_{\mathsf{vsf}}$ and $\mathsf{DTMFA}_{\mathsf{mcf}}$ that works in PSPACE (enumerate all smaller candidates and check equivalence). We leave open whether this is optimal, but observe that even for DREG, minimization is NP-complete, see Niewerth [42].

Next, we discuss a potential extension of determinism. One could argue that Definition 14 is overly restrictive; e. g., consider $\alpha := \langle x\colon \mathtt{a}^+\rangle\langle y\colon \mathtt{b}+\rangle\mathtt{c}(\&x \vee \&y)$. Then α is not deterministic; but as the contents of x and y always start with a or b (respectively), deterministic choices between $\&x$ and $\&y$ are possible by looking at the current letter of the input word. Analogous observations can be made for TMFA. More precisely, we define the notion of *ℓ-deterministic* TMFA as a relaxation of the criteria of DTMFA: In contrast to the latter, an ℓ-deterministic TMFA can have states q with multiple memory recall-transitions, as long as these recall distinct memories, and if q is reached in some computation, then for each pair of these recalled memories, the contents differ in the first ℓ positions. First, note that this does not increase the expressive power (intuitively, storing the length ℓ prefixes of the memory contents allows making ℓ-deterministic memory recall transitions deterministic):

▶ **Proposition 31.** *Let $\ell \geq 1$. For every ℓ-deterministic $M \in$ DTMFA, there is an $M' \in$ DTMFA with $\mathcal{L}(M) = \mathcal{L}(M')$.*

For the sake of the argument, let $\alpha \in$ RX be ℓ-deterministic if and only if $\mathcal{M}(\alpha)$ is.

▶ **Proposition 32.** *For every $\ell \geq 1$, deciding whether a TMFA is ℓ-deterministic is PSPACE-complete. The problem is coNP-complete if the input is restricted to $\mathsf{TMFA}_{\mathsf{mcf}}$. These lower bounds hold even if we restrict the input to RX and $\mathsf{RX}_{\mathsf{vsf}}$, respectively.*

Hence, while we can decide efficiently whether a TMFA or a regex is deterministic, detecting ℓ-determinism is costly, even for $\ell = 1$. The same holds if we adapt the definition to distinguish between variables and terminals (see Section A.21 in the full version of the paper [26]).

7 Conclusions and Further Directions

Based on TMFA, an automaton model for regex, we extended the notion of determinism from regular expressions to regex. Although the resulting language class cannot express all regular languages, it is still rich; and by using a generalization of the Glushkov construction, deterministic regex can be converted into a DTMFA, and the membership problem can then be solved quite efficiently. Although we did not discuss this, the construction is also compatible with the Glushkov construction with counters by Gelade, Gyssens, Martens [29]. Hence, one can add counters to DRX and DTMFA without affecting the complexity of membership.

Many challenging questions remain open, for example: Can the more advanced results for DREG be adapted to DRX, i. e., can $\mathcal{M}(\alpha)$ be computed more efficiently (as in [8, 43]), or is it even possible, like in [31], to avoid computing $\mathcal{M}(\alpha)$? Is effective minimization possible for DTMFA or DRX? Is it decidable whether a DTMFA defines a DRX-language? Are inclusion and equivalence decidable for DRX or DTMFA? Can determinism be generalized to larger classes of regex without making the membership problem intractable?

Acknowledgements. The authors thank Wim Martens for helpful feedback, Matthias Niewerth, for pointing out that v_n must be a factor of p_n in the jumping lemma, and Martin Braun, for creating a library and tool for DRX and DTMFA (available at [6]).

References

1 Abigail. Re: Random number in perl. Posting in the newsgroup comp.lang.perl.misc, October 1997. Message-ID slrn64sudh.qp.abigail@betelgeuse.wayne.fnx.com.

2 Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. Elsevier, Amsterdam, 1990.

3 Dana Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21:46–62, 1980.

4 Pablo Barceló, Carlos A. Hurtado, Leonid Libkin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. In *Proc. PODS 2010*, 2010.

5 Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Trans. Web*, 4(4):14, 2010.

6 Martin Braun. moar – Deterministic Regular Expressions with Backreferences, 2016. Accessed December 2016. URL: `https://github.com/s4ke/moar`.

7 Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language XML 1.0 (fifth edition). W3C recommendation. Technical Report `https://www.w3.org/TR/2008/REC-xml-20081126/`, W3C, November 2008.

8 Anne Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.

9 Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Inf. Comput.*, 142(2):182–206, 1998.

10 Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *Int. J. Found. Comput. Sci.*, 14:1007–1018, 2003.

11 Benjamin Carle and Paliath Narendran. On extended regular expressions. In *Proc. LATA 2009*, 2009.

12 Pascal Caron, Yo-Sub Han, and Ludovic Mignot. Generalized one-unambiguity. In *Proc. DLT 2011*, pages 129–140, 2011.

13 Marek Chrobak. Finite automata and unary languages. *Theor. Comput. Sci.*, 47(3):149–158, 1986.

14 Marek Chrobak. Errata to: "Finite automata and unary languages". *Theor. Comput. Sci.*, 302(1):497–498, 2003.

15 Russ Cox. Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). `https://swtch.com/~rsc/regexp/regexp1.html`, 2007. Accessed: 2016-09-03.

16 Wojciech Czerwinski, Claire David, Katja Losemann, and Wim Martens. Deciding definability by deterministic regular expressions. In *Proc. FOSSACS 2013*, pages 289–304, 2013.

17 Volker Diekert. Makanin's Algorithm. In *Algebraic Combinatorics on Words* [38], chapter 12.

18 Volker Diekert. More than 1700 years of word equations. In *Proc. CAI 2015*, 2015.

19 Volker Diekert, Artur Jeż, and Wojciech Plandowski. Finding all solutions of equations in free groups and monoids with involution. In *Proc. CSR 2014*, 2014.

20 Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12, 2015.

21 Henning Fernau and Markus L. Schmid. Pattern matching with variables: A multivariate complexity analysis. *Inform. Comput.*, 242:287–305, 2015.

22 Henning Fernau, Markus L. Schmid, and Yngve Villanger. On the parameterised complexity of string morphism problems. *Theory Comput. Syst.*, 59(1):24–51, 2016.

23 Dominik D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory Comput. Sys.*, 53(2):159–193, 2013.

24 Dominik D. Freydenberger. A logic for document spanners. In *Proc. ICDT 2017*, 2017. Accepted. Available at `http://ddfy.de/publications/F-ALfDS.html`.

25 Dominik D. Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. In *Proc. ICDT 2016*, 2016.

26 Dominik D. Freydenberger and Markus L. Schmid. Deterministic regular expressions with back-references. Full version of this paper. URL: `http://ddfy.de/publications/FS-DREwBR.html`.

27 Shudi (Sandy) Gao, C. M. Sperberg-McQueen, and Henry S. Thompson. W3C XML schema definition language (XSD) 1.1 part 1: Structures. Technical Report `https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/`, W3C, April 2012.

28 Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

29 Wouter Gelade, Marc Gyssens, and Wim Martens. Regular expressions with counting: Weak versus strong determinism. *SIAM J. Comput.*, 41(1):160–190, 2012.

30 Jan Goyvaerts. Regular expressions tutorial. `http://www.regular-expressions.info/tutorial.html`, 2016. Accessed December 2016.

31 Benoît Groz, Sebastian Maneth, and Slawek Staworko. Deterministic regular expressions in linear time. In *Proc. PODS 2012*, 2012.

32 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

33 Michael Kay. XPath and XQuery Functions and Operators 3.0 W3C Recommendation. Technical Report `https://www.w3.org/TR/2014/REC-xpath-functions-30-20140408/`, W3C, April 2014.

34 S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon, J. McCarthy, and W. R. Ashby, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, NJ, 1956.

35 Dexter Kozen. Lower bounds for natural proof systems. In *Proc. FOCS 1977*, 1977.

36 Markus Latte and Matthias Niewerth. Definability by weakly deterministic regular expressions with counters is decidable. In *Proc. MFCS 2015*, 2015.

37 Katja Losemann, Wim Martens, and Matthias Niewerth. Closure properties and descriptional complexity of deterministic regular expressions. *Theor. Comput. Sci.*, 627:54–70, 2016.

38 M. Lothaire. *Algebraic Combinatorics on Words*, volume 90 of *Encyclopedia of mathematics and its applications.* Cambridge University Press, 2002.

39 Ping Lu, Joachim Bremer, and Haiming Chen. Deciding determinism of regular languages. *Theory Comput. Syst.*, 57(1):97–139, 2015.

40 Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM J. Comput.*, 39(4):1486–1530, 2009.

41 Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM TOIT*, 5(4):660–704, 2005.

42 Matthias Niewerth. *Data Definition Languages for XML Repository Management Systems.* PhD thesis, TU Dortmund, 2015. URL: `http://www.theoinf.uni-bayreuth.de/en/downloads/PHD_Niewerth.pdf`.

43 Jean-Luc Ponty, Djelloul Ziadi, and Jean-Marc Champarnaud. A new quadratic algorithm to convert a regular expression into an automaton. In *Proc. WIA '96*, 1996.

44 Markus L. Schmid. Inside the class of regex languages. *Int. J. Found. Comput. Sci.*, 24(7):1117–1134, 2013.

45 Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inform. Comput.*, 249:1–17, 2016.

A Appendix

A.1 Regex in Theory and Practice

In this section, we motivate the choice of the formalization of regex syntax and semantics that are used in the current paper, in particular in comparison to [10], and then connect these to the use of back-references in actual implementations.

Choices behind the definition: We begin with a discussion of semantics of back-references, which most actual implementations define[2] in terms of the used matching algorithm. For a theoretical analysis, this approach is not satisfactory. Câmpeanu, Salomaa, Yu [10] then proposed a definition using parse trees, which was precise, but rather technical and unwieldy. Schmid [45] then introduced the definition with ref-words that we use in the current paper. Both definitions are equivalent, with a caveat that we discuss below.

The most obvious difference in approaches to syntax is that some formalizations, like [10], do not use variables, but numbered back-references. For example, $\langle x\colon \mathtt{a}^*\rangle\mathtt{b}\cdot \& x$ would be written as $(\mathtt{a}^*)\mathtt{b}\backslash 1$, where $\backslash 1$ refers to the content of the first pair of parentheses (called the first *group*).

After working with this definition for some time, the authors of the present paper came to the conclusion that using numbered back-references instead of named variables is inconvenient (both when reading and writing regex). The developers of actual implementations seem to agree with this sentiment: While using numbered back-references was well-motivated when considering PERL at the time [10] was published, most current regex dialects allow the use of *named groups*, which basically act like our variables (depending on the actual dialect, see below). The choice between variables and numbered groups is independent of the choice of semantics, as parse trees can also be used with variables, see [23]. Hence, using variables instead of numbers is a natural choice.

Building on this, the next question is whether the same variable can be bound at different places in the regex, i. e., whether one allows expressions like

$$\big(\langle x\colon \mathtt{a}^+\rangle\langle y\colon \mathtt{b}^+\rangle\big)\vee\big(\langle y\colon \mathtt{b}^+\rangle\langle x\colon \mathtt{a}^+\rangle\big)\mathtt{c}\cdot \& x\cdot \& y.$$

While some implementations that have developed from back-references forbid theses constructions to certain degrees (again, see below), there seems to be no particular reason for this decision when approaching this question without this historical baggage. In fact, one can argue from a point of applications that expressions like the following make sense (abstracting away some details that would be needed in actual use):

$$\Sigma^*\Big(\big(\mathtt{Name:}\langle x\colon \Sigma^+\rangle\ \mathtt{Title:}\langle y\colon \Sigma^+\rangle\big)\vee\big(\mathtt{Title:}\langle y\colon \Sigma^+\rangle\ \mathtt{Name:}\langle x\colon \Sigma^+\rangle\big)\Big)\Sigma^*$$

In fact, these constructions are explicitly allowed in the regex formulas of Fagin et al. [20], that are closely related to regex. In particular, both the semantic definitions (ref-words and parse trees) allow this choice. Thus, there seems to be no particular practical reason to disallow these constructions when considering only the model (instead of its algorithmic properties).

The regex definition in [10] also includes a syntactic restriction that changes the expressive power considerably: It requires that a backreference $\backslash n$ can only appear in a regex if it occurs

[2] From a theory point of view, this is a generous use of the term "define".

to the right of corresponding group number n. In [10], otherwise, the expression is called a "semi-regex". Consider $\alpha_{\mathsf{sq}} = \big(\langle x\colon \& y\rangle\langle y\colon \& x \cdot \mathtt{a}\rangle\big)^*$ from Example 2. In the numbered notation of [10], this would be expressed as $\beta := \big(({}_2\backslash 3)_2({}_3\backslash 2 \cdot \mathtt{a})_3\big)^*$, when adding group numbers to the groups to increase readability. But using definitions from [10], β is only a semi-regex, as the reference \3 occurs to the left of group 3.

The motivation behind this restriction is not explained in [10]. While one might argue that this was chosen to avoid referencing unused groups, the definition of semantics in [10] still needs to deal with this problem in regexes like $(({}_1\mathtt{a})_1 \vee ({}_2\mathtt{a})_2)\cdot\backslash 1\cdot\backslash 2$, and handles them by assigning ε (like the definition from [45], which we use as well). Hence, even on "semi-regex", the parse tree semantics behave like the ref-word semantics.

Arguably, the restriction has an advantage from a theoretical point of view, as it allows Câmpeanu, Salomaa, Yu [10] and Carle and Narendran [11] to define pumping lemmas for this class. Using these, it is possible to show that languages like $\mathcal{L}(\alpha_{\mathsf{sq}})$ from Example 2 or the language from Example 20 cannot be expressed with the regex model from [10]. But in other areas, there seems to be no advantage in this choice: Even under this restriction, the membership problem is NP-complete (since it is still possible to describe Angluin's pattern languages [3]), the undecidability results from [23] on various problems of static analysis are unaffected by this choice, and even the proof of Theorem 28 directly works on this subclass. In summary, the authors of the current paper see no reason to adapt this restriction.

For full disclosure, the second author points out that he misinterpreted the regex definition of [10] when citing the paper in his own articles [44, 45]. All mentions of "regex" in those papers refer to the expressions that are called "semi-regex" in [10]. Hence, although those papers refer to [10] for the full definition of regex, they talk about the language class $\mathcal{L}(\mathsf{RX})$ of the current paper.

The last choice in the definition that we need to address is how we deal with referencing undefined variables. Both [10] and [45] default those references to ε (as do others, like [23]); but there is also literature, like [11], that uses $\emptyset$ as default value (under these semantics, a ref-word that contains a variable that de-references to $\emptyset$ cannot generate any terminal words; the same holds for a parse tree that contains such a reference). This choice can easily be implemented in both semantics by discarding a ref-word or parse tree that contains such a reference; and a TMFA can reject if a run encounters a reference to such an undefined memory.

While these "$\emptyset$-semantics" are also used in some actual implementations, the authors of the current paper are against this approach. One of the reasons is that using $\emptyset$ as default allows the use of curious synchronization effects that distract from the main message of this paper. For example, let $\Sigma = \{a_1, \ldots, a_n\}$ for some $n \geq 1$, and define

$$\alpha_n := \Big(\bigvee_{i=1}^{n} (a_i \cdot \langle x_i \colon \varepsilon\rangle)\Big)^n \cdot \& x_1 \cdots \& x_n.$$

If unbound variables default to $\emptyset$, this regex generates the language

$$\big\{a_{\pi(1)} \cdots a_{\pi(n)} \mid \pi \text{ is a permutation of } \{1, \ldots, n\}\big\},$$

as every variable x_i needs to be assigned ε exactly once (otherwise, a reference would return $\emptyset$ and block). Hence, using this semantics, even variables that are bound only to ε can be used for synchronization effects. While this can lead to interesting constructions, the authors think that it provides more insight to study the effects of back-references on lower bounds without relying on these additional features. This way, there is no question whether the hardness of the examined problems is due to the effects of the $\emptyset$-semantics.

Furthermore, all examples in the present paper can be adapted from the used ε-semantics to $\emptyset$-semantics: Given an $\alpha \in \mathsf{RX}$ with variables $x_1, \ldots, x_k$, define $\alpha' := \langle x_1 : \varepsilon \rangle \cdots \langle x_k : \varepsilon \rangle \cdot \alpha$. First, we observe that the language that is defined by α' under $\emptyset$-semantics is the same language that α defines under ε-semantics. Furthermore, note that if α is deterministic, α' is also deterministic. The analogous construction can be used for TMFA (and DTMFA).

While it is possible to adapt most of the results in the current paper directly to this alternative semantics, the authors chose to keep the paper focused on ε-semantics.

Actual implementations: We now give a brief overview of how back-references are used in some actual implementations. For a good introduction on various dialects, the authors recommend [30], in particular the section on back-references and named groups. As this behavior is often under-defined, badly documented, and implementation dependent, this can only be a very short and superficial summary of some behavior.

Before we go into details, we address why back-references are used, in spite of the resulting NP-hard membership problem: Most regex libraries use a backtracking algorithm that can have exponential running time, even on many proper regular expressions (see Cox [15]). From this point of view, back-references can be added with little implementation effort and without changing the efficiency of the program.

Most modern dialects of regex not only support numbered back-references as used by [10], but also named capture groups, which work like our variables. In some dialects like e. g. Python, PERL, and PCRE, these act as aliases for back-references with numbers; hence, $\langle x\colon \mathtt{a}^*\rangle\mathtt{b}\& x$ would be interpreted as $(\mathtt{a}^*)\mathtt{b}\backslash 1$. As a consequence, each name resolves to a well-defined number. As some of these dialects assign the empty set as default value of unbound back-references (or group names), the resulting behavior is similar implicitly requiring the restriction from [10]. This implementation of named capture groups seems to be mostly for historical reasons (as back-references were introduced earlier).

In contrast to this, there are other dialects that use numbered back-references and explicitly allow references to access groups that occur to their right in the expression. For example, the W3C recommendation for XPath and XQuery functions and operators [33] defines regular expressions with back-references for the use in `fn:matches`. There, it is possible to refer to capture groups that occur to the right of the reference (although only for the capture groups 1 to 9, but not for 10 to 99, which might be considered a peculiar decision). As this dialect defaults unbound references to ε, it is possible to directly express α_{sq} by renaming the variable references to back-references.

Furthermore, .NET allows names to be used for many different groups, for example $((\langle x\colon \mathtt{a}^+\rangle \vee \langle x\colon \mathtt{b}^+\rangle\mathtt{c})\& x)^*$. While .NET defaults unset variables to $\emptyset$, it is possible to express $\mathcal{L}(\alpha_{\mathsf{sq}})$, by using an expression like $\langle x\colon \varepsilon\rangle \cdot \alpha_{\mathsf{sq}}$. In the same way, every regex in the sense of our paper can be converted into an equivalent .NET regex.

Finally, in 2007 (just four years after the publication of [10]), PERL 5.10 introduced *branch reset groups* (which were also adapted in PCRE). These reset the numbering inside disjunctions, and allow expressions that behave like $((\langle x\colon \mathtt{a}^+\rangle \vee \langle x\colon \mathtt{b}^+\rangle\mathtt{c})\& x)^*$. This allows PERL regex to replicate a large part of the behavior of .NET regex.

In conclusion, it seems that almost every formalization of regex syntax and semantics can be justified by finding the right dialect; but every restriction might be superseded by the continual evolution of regex dialects. Hence, the current paper attempts to avoid restrictions; and when in doubt, we choose natural definitions over trying to directly emulate a single dialect. Therefore, we use variables instead of numbered back-references, and allow multiple uses of the same variable name.

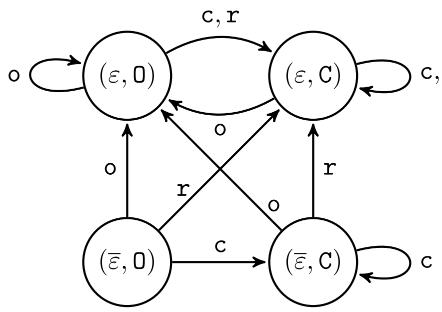


Figure 2 Possible configuration changes of a fixed memory.

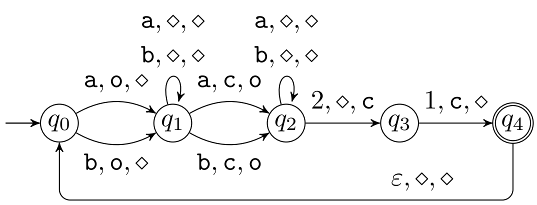
The authors acknowledge that most actual implementations of "regular expressions" allow additional operators. Common features are *counters*, which allow constructions like e. g. $\mathtt{a}^{2,5}$ that define the language $\{\mathtt{a}^i \mid i \in \{2, \ldots, 5\}\}$, *character classes* and *ranges*, which are shortcuts for sets like "all alphanumeric symbols" or "all letters from b to y", and *look ahead* and *look behind*, which can be understood as allowing the expression to call another expressions as a kind of subroutine.

While these operators are outside of the scope of the current paper, we briefly address the issue of counters. These are used in XML DTDs and XML Schema, and were studied in connection to determinism. In particular, Gelade, Gyssens, Martens [29] described how counters can be added to finite automata and proposed an appropriate extension of determinism and Glushkov construction to this model. Although the current paper does not address this matter (in order to keep the paper focussed), the TMFA that we introduce in Section 3 can also be extended with counters (like the extension to NFA in [29]). Likewise, the Glushkov constructions of [29] and the current paper can be combined, as can the notions of determinism. The membership problem for the resulting class of deterministic regex with counters can then be solved as efficiently as for deterministic regex (see Theorem 19).

A.2 Examples and Illustrations for Memory Automata with Trap State

Intuitively speaking, in a single step of a computation of a TMFA, we first change the memory statuses according to the memory instructions s_i, $1 \leq i \leq k$, and then a (possibly empty) prefix v of the remaining input (v is either from $\Sigma \cup \{\varepsilon\}$ or it equals the content of some memory that, according to the definition, has been closed by the same transition) is consumed and appended to the content of every memory that is currently open (note that here the new statuses after applying the memory instructions count). The changes of memory configurations caused by a transition are illustrated in Figure 2 (by ε and $\overline{\varepsilon}$, we denote an empty or non-empty memory content, respectively; the instruction $\diamond$ is omitted).

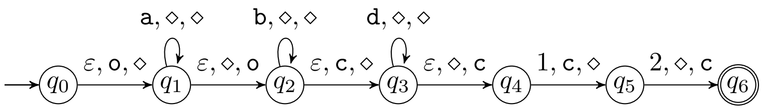
▶ **Example 33.** Consider the following $\mathsf{TMFA}^{\mathsf{rej}}$ M with two memories over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$:



This TMFA works as follows. First, we record a non-empty word over $\{\mathtt{a},\mathtt{b}\}$ in the first memory, then a non-empty word over $\{\mathtt{a},\mathtt{b}\}$ in the second memory, and then these words are repeated in reverse order by first recalling the second and then the first memory (note that in the transition from q_3 to q_4, an already closed memory is closed again, since according to Definition 3, every memory that is recalled must be closed in the same transition). Due to the ε-transition from q_4 to q_0, M describes the Kleene-plus of such words, i. e., $L(M) = L(\alpha)$, where $\alpha = (\langle x\colon (\mathtt{a} \vee \mathtt{b})^+\rangle\langle y\colon (\mathtt{a} \vee \mathtt{b})^+\rangle \cdot \&y \cdot \&x)^+ = (\{uvvu \mid u, v \in \{\mathtt{a},\mathtt{b}\}^+\})^+$.

Note that each of the two memory recall transitions closes the respective memory. This is required by definition, as a transition can only recall a memory if it ensures that it is closed.

▶ **Example 34.** Consider the following TMFA$^{\mathsf{rej}}$ M with two memories over $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{d}\}$:



The behavior of M can be described as follows: First, M opens memory 1 and reads $\mathtt{a}^i$, $i \geq 0$. After that, M opens the second memory, reads $\mathtt{b}^j$, $j \geq 0$, closes the first memory, reads $\mathtt{d}^k$, $k \geq 0$, and closes the second memory. Hence, after reading $\mathtt{a}^i\mathtt{b}^j\mathtt{d}^k$, the first memory contains $\mathtt{a}^i\mathtt{b}^j$, and the second $\mathtt{b}^j\mathtt{d}^k$. Finally, M recalls memory 1 and then 2. Hence, $\mathcal{L}(M) = \{\mathtt{a}^i\mathtt{b}^j\mathtt{d}^k\mathtt{a}^i\mathtt{b}^j\mathtt{b}^j\mathtt{d}^k \mid i,j,k \geq 0\}$.

Now, note that in each input word w, memory 2 is opened and closed after memory 1. Hence, if $j > 0$, the areas in w where the two memories are open overlap, instead of being nested. This cannot happen in a regex, as it is ensured from the syntax of variable bindings that these "areas" in the word are properly nested. For this reason, it seems impossible to express $\mathcal{L}(M)$ with a regex with two variables. But this does not mean that $\mathcal{L}(M)$ is not a regex language, as $\mathcal{L}(M) = \mathcal{L}(\alpha)$ for $\alpha = \langle x\colon \mathtt{a}^*\rangle\langle y\colon \mathtt{b}^*\rangle\langle z\colon \mathtt{d}^*\rangle \cdot \&x\&y \cdot \&y\&z$. In other words, the key idea is expressing each memory with two variables (one for the overlapping parts of the memories, and one for each rest).

Clearly, the TMFA of Examples 33 and 34 are not deterministic (in Example 33, there are different transitions for the same state that consume the same symbol and in Example 34, there are states for which ε-transitions exists in addition to other transitions). By minor changes of the TMFA of Example 33, a DTMFA can be easily constructed for the language $(\{u\mathtt{d}v\mathtt{d}vu \mid u, v \in \{\mathtt{a},\mathtt{b}\}^+\})^+$, the details are left to the reader.

A.3 Proof of Theorem 4

We first need the following definition.

An $M \in$ TMFA is in *normal form* if no empty memory is recalled, no open memory is opened, no memory is reset, and, for every transition $\delta(q, b) \ni (p, s_1, \ldots, s_k)$,

- if $b \neq \varepsilon$, then $s_i = \diamond$, $1 \leq i \leq k$,
- if $s_i \neq \diamond$, for some i, $1 \leq i \leq k$, then $b = \varepsilon$ and $s_j = \diamond$, for every j, $1 \leq j \leq k$, $i \neq j$.

▶ **Proposition 35.** *Any* TMFA *can be transformed into an equivalent* TMFA *in normal form.*

Proof. An arbitrary TMFA can be changed into an equivalent one in normal form as follows. By introducing ε-transitions, we can make sure that every transition is of the form stated in the proposition. Furthermore, by adding states, we can keep track of the memory configurations (i. e., their status and whether or not they are empty (see proof of Theorem 6)). This allows us to replace transitions that are recalling an empty memory by ε-transitions.

Furthermore, transitions that open an open memory i are replaced by transitions applying the memory instructions c and o in this order to memory i, and transitions that reset a memory i are replaced by transitions applying the memory instructions c, o and c in this order to memory i. The TMFA is then in normal form and, by definition, these modifications do not change the accepted language. ◀

Now, we can state the proof of Theorem 4.

Proof. We first note that $\mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}}) = \mathcal{L}(\mathsf{RX})$ follows from [45] (we briefly discuss this at the end of the proof). Since $\mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}}) \subseteq \mathcal{L}(\mathsf{TMFA})$ and $\mathsf{TMFA} = \mathsf{TMFA}^{\mathsf{rej}} \cup \mathsf{TMFA}^{\mathsf{acc}}$, it only remains to prove $\mathcal{L}(\mathsf{TMFA}^{\mathsf{acc}}) \subseteq \mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}})$. To this end, let M be a $\mathsf{TMFA}^{\mathsf{acc}}$ in normal form. First, we replace every memory i, $1 \leq i \leq k$, by two memories $(i, 1)$ and $(i, 2)$ and we implement in the finite state control a list $(x_1, x_2, \dots, x_k)$ with entries from $\Sigma \cup \{\varepsilon\}$, which initially satisfies $x_i = \varepsilon$, $1 \leq i \leq k$. Then, we change the transitions of M such that the new memories $(i, 1)$ and $(i, 2)$ simulate the old memory i, i. e., memory i stores some word u if and only if memories $(i, 1)$ and $(i, 2)$ store u_1 and u_2, respectively, with $u = u_1u_2$. Moreover, the element x_i always equals the first symbol of the content of memory $(i, 2)$. More precisely, this can be done as follows. Let $\delta(q, b) \ni (p, s_1, \dots, s_k)$ be an original transition of M.

- If $s_i = \mathsf{o}$ or $s_i = \mathsf{c}$, for some i, $1 \leq i \leq k$, then instead we open memory $(i, 1)$ or close memory $(i, 2)$, respectively.
- If $b \in \Sigma$, then, for every open memory $(i, 1)$, we nondeterministically choose to close it and open memory $(i, 2)$ instead and set $x_i = b$. Then we read b from the input and change to state p.
- If $b \in \{1, 2, \dots, k\}$, then we first recall memory $(b, 1)$ and then, for every open memory $(i, 1)$, we nondeterministically choose to close it and open memory $(i, 2)$ instead and set $x_i = x_b$. Then, we recall memory $(b, 2)$ and change to state p.

All these modifications can be done by introducing intermediate states and using ε-transitions and the accepted language of M does not change.

The automaton M now stores some content u of an original memory i factorised into two factors u_1 and u_2 in the memories $(i, 1)$ and $(i, 2)$, respectively. For the sake of convenience, we simply say that u is stored in $(i, 1) \cdot (i, 2)$ in order to describe this situation. Next, we show that if u is stored in $(i, 1) \cdot (i, 2)$, then any way of how u is factorised into the content of $(i, 1)$ and $(i, 2)$ is possible. More precisely, we show that, for every $w, u, u_1, u_2 \in \Sigma^*$ with $u_1u_2 = u$, M can reach state p by consuming w with u stored in $(i, 1) \cdot (i, 2)$ if and only if M can reach state p by consuming w with u_1 and u_2 stored in $(i, 1)$ and $(i, 2)$, respectively .

The *if* part of this statement is trivial. We now assume that M can reach state p by consuming w with u stored in $(i, 1) \cdot (i, 2)$. This implies that we reach the situation that $(i, 1)$ is open, currently stores u'_1 and the next consuming transition consumes $u''_1u'_2$, where $u_1 = u'_1u''_1$ and $u_2 = u'_2u''_2$ with $u'_2 \neq \varepsilon$. If $u''_1 = \varepsilon$, then M can choose to close $(i, 1)$ and then open $(i, 2)$, which results in u_1 and u_2 being stored in $(i, 1)$ and $(i, 2)$, respectively. If, on the other hand, $u''_1 \neq \varepsilon$, then the next transition recalls memories $(j, 1)$ and $(j, 2)$ such that $u''_1u'_2$ is stored in $(j, 1) \cdot (j, 2)$. If u''_1 and u'_2 are stored in $(j, 1)$ and $(j, 2)$, respectively, then M first recalls $(j, 1)$, chooses to close $(i, 1)$ and open $(i, 2)$, and then recalls $(j, 2)$, which results in u_1 and u_2 being stored in $(i, 1)$ and $(i, 2)$. Consequently, we have to repeat this argument for memories $(j, 1)$ and $(j, 2)$, i. e., we have to show that it is possible that $u''_1u'_2$ is stored in $(j, 1) \cdot (j, 2)$ in such a way that u''_1 is stored in $(j, 1)$ and u'_2 is stored in $(j, 2)$. Repeating this argument, we will eventually arrive at a memory that is not filled by any memory recalls; thus, we necessarily have the case $u''_1 = \varepsilon$.

Now, we turn M into a $\mathsf{TMFA}^{\mathsf{rej}}$ M', i. e., the state [trap] becomes non-accepting, and, in addition, we add a new new accepting state q_t (simulating the old accepting [trap]) with $\delta(q_t, x) = \{(q_t, \diamond, \ldots, \diamond)\}$, $x \in \Sigma$, and have all transitions of the former M that lead to [trap] (note that these are not memory recall failure transitions) now leading to q_t. Furthermore, we change this M' such that for every memory recall, there is also the nondeterministic choice to only recall $(i, 1)$, then check whether x_i does not equal the next symbol on the input and, if this is the case, enter state q_t. Obviously, this simulates the memory recall failure of M.

Every word accepted by M without memory recall failures can be accepted by M' in the same way, every word accepted by M due to a recall failure can be accepted by M' by guessing and simulating this memory recall failure. On the other hand, if M' accepts a word with a simulated memory recall failure, then M will accept this word by a proper memory recall failure, and if M' accepts a word without a simulated memory recall failure, then, since $M' \in \mathsf{TMFA}^{\mathsf{rej}}$, there is no memory recall failure in the computation and M can accept the word by the same computation.

This completes the proof of $\mathcal{L}(\mathsf{TMFA}^{\mathsf{acc}}) \subseteq \mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}})$.

We shall conclude this proof by briefly sketching why $\mathcal{L}(\mathsf{TMFA}^{\mathsf{rej}}) = \mathcal{L}(\mathsf{RX})$ holds. For an $\alpha \in \mathsf{RX}$, it is straightforward to obtain an equivalent $\mathsf{TMFA}^{\mathsf{rej}}$: Transform α into a proper regular expression $\alpha_{\mathcal{R}}$ with $\mathcal{L}(\alpha_{\mathcal{R}}) = \mathcal{R}(\alpha)$ (by just renaming variable bindings and references), then transform $\alpha_{\mathcal{R}}$ into an equivalent NFA M, and finally interpret M as a $\mathsf{TMFA}^{\mathsf{rej}}$ by interpreting transition labels $[_x$, $]_x$ as memory instructions and transition labels x as memory recalls. The other direction relies on first resolving overlaps of memories (i. e., the case that two memories store factors that overlap in the input word, see also Example 34) and then transforming the $\mathsf{TMFA}^{\mathsf{rej}}$ M into a proper regular expression for a ref-language that dereferences to $\mathcal{L}(M)$, which can then directly be interpreted as a regex (due to the non-overlapping property of memories, which translates into a well-formed nesting of the parentheses $[_x$, $]_x$). The details here are a bit more technical and we refer to [45] for further reading. ◀

A.4 Proof of Theorem 6

We first extend the notion of completeness from DFA to DTMFA, by saying that a DTMFA is *complete* if, for every $q \in Q$, either $\delta(q, x)$ is defined, for every $x \in \Sigma$, or $\delta(q, i)$ is defined, for some i, $1 \leq i \leq k$, or $\delta(q, \varepsilon)$ is defined. This means that a complete DTMFA has, for every state, either exactly $|\Sigma|$ transitions (which are all consuming transitions, but not memory recall transitions), exactly one memory recall transition, or exactly one ε-transition.

For deterministic automata, it is usually possible to apply the state complementation technique (i. e., toggling acceptance of states) in order to show closure under complement. However, we also need completeness and ε-freeness, since otherwise it may happen that a word is not accepted because its computation gets stuck or enters an infinite ε-loop and therefore is not entirely processed, which leads to a word which is accepted neither by the original nor by the complement automaton. The requirement of completeness and ε-freeness is not a restriction for DTMFA, since these properties can be achieved by classical techniques. However, recalling empty memories, which are special cases of ε-transition, can cause the same problems and therefore we have to get rid of them as well. This can be done by storing in the finite-state control whether the memories are currently empty or non-empty and then treating recalls of empty memories as ε-transitions and remove them along with the other ε-transition in the classical way.

We need a few more definitions: Let $\Gamma = \{\mathtt{o}, \mathtt{c}, \mathtt{r}, \diamond\}$ and let $\circledcirc$ be a binary operator on Γ defined by $x \circledcirc y = y$, if $y \neq \diamond$ and $x \circledcirc y = x$, if $y = \diamond$. Furthermore, we extend $\circledcirc$ to Γ^k by

$(x_1, \ldots, x_k) \circledcirc (y_1, \ldots, y_k) = (x_1 \circledcirc y_1, \ldots, x_k \circledcirc y_k)$. We note that $\circledcirc$ is associative and some memory instructions $s_1, s_2, \ldots, s_n \in \Gamma$ applied to some memory in this order have the same result as the memory instruction $s_1 \circledcirc s_2 \circledcirc \ldots \circledcirc s_n$.

Next, we prove a sequence of propositions:

▶ **Proposition 36.** *Let $M \in$ DTMFA. For every $w \in \Sigma^*$ and every configuration c for M, there exists at most one configuration c' with $c \vdash_M c'$.*

Proof. Let $c = (q, v, (u_1, r_1), \ldots, (u_k, r_k))$. If no $\delta(q, i)$, $1 \leq i \leq k$, is defined, then there is obviously at most one c' with $c \vdash_m c'$. If $\delta(q, i) = (p, s_1, \ldots, s_k)$, for some i, $1 \leq i \leq k$, then either $v = u_i v'$, which implies that $c \vdash_M (p, v', (u'_1, r'_1), \ldots, (u'_k, r'_k))$, where the (u'_j, r'_j), $1 \leq j \leq k$, are uniquely determined by u_i and the s_j, $1 \leq j \leq k$, or u_i is not a prefix of v, which implies that $c \vdash_M ([\mathsf{trap}], v'', (u_1, r_1), \ldots, (u_k, r_k))$, where $v = v'v''$ and v' is the largest common prefix of v and u_i. In both cases, there is at most one configuration c' with $c \vdash_M c'$. ◀

▶ **Proposition 37.** *For every $M \in$ DTMFA there exists an ε-free $M' \in$ DTMFA with $\mathcal{L}(M) = \mathcal{L}(M')$.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$. For every $p \in Q$, if, for some $q \in Q$, $\delta(p, \varepsilon) = (q, s_1, \ldots, s_k)$, then we define $\mathcal{S}_{\varepsilon,1}(p) = q$ and $\mathcal{M}(p, q) = (s_1, \ldots, s_k)$. For every $p \in Q$ and every i, $2 \leq i \leq |Q| - 1$, we define $\mathcal{S}_{\varepsilon,i}(p) = \mathcal{S}_{\varepsilon,1}(\mathcal{S}_{\varepsilon,i-1}(p))$ and, if $\mathcal{S}_{\varepsilon,i}(p)$ is defined, we define (or redefine) $\mathcal{M}(p, \mathcal{S}_{\varepsilon,i}(p)) = \mathcal{M}(p, \mathcal{S}_{\varepsilon,i-1}(p)) \circledcirc (s_1, \ldots, s_k)$, where $\delta(\mathcal{S}_{\varepsilon,i-1}(p), \varepsilon) = (\mathcal{S}_{\varepsilon,i}(p), s_1, \ldots, s_k)$.

For every $p \in Q$ with $\delta(p, \varepsilon)$ defined, we now remove the ε-transitions as follows. Let i, $1 \leq i \leq |Q| - 1$, be such that $\mathcal{S}_{\varepsilon,i}(p) = q$ and $\mathcal{S}_{\varepsilon,i+1}(p)$ is undefined. Furthermore, let $\delta(q, x_j) = (t_j, s_{j,1}, \ldots, s_{j,k})$, $1 \leq j \leq \ell$, for some ℓ with $0 \leq \ell \leq |\Sigma|$, be all the transitions from q (note that $\ell = 1$ and $x_1 \in \{1, 2, \ldots, k\}$ covers the case of a single memory recall transition and, furthermore, an ε-transition is not possible). We now add new transitions $\delta(p, x_j) = (t_j, s'_1 \circledcirc s_{j,1}, \ldots, s'_k \circledcirc s_{j,k})$, where $\mathcal{M}(p, q) = (s'_1, \ldots, s'_k)$. Then, we simply delete all ε-transitions (note that this may produce states that are not reachable anymore, which are deleted as well). It can be easily verified that this results in an $M' \in$ DTMFA with $\mathcal{L}(M) = \mathcal{L}(M')$. ◀

▶ **Proposition 38.** *For every $M \in$ DTMFA there exists a complete $M' \in$ DTMFA with $\mathcal{L}(M) = \mathcal{L}(M')$.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$. We transform M into M' by adding a new non-accepting state t with $\delta(t, x) = (t, \diamond, \ldots, \diamond)$, for every $x \in \Sigma$, and we add transitions for every state $q \in Q$ as follows. If $\delta(q, i)$ is undefined, for every i, $1 \leq i \leq k$, and $\delta(q, \varepsilon)$ is undefined, then, for every $x \in \Sigma$ with $\delta(q, x)$ undefined, we set $\delta(q, x) = (t, \diamond, \ldots, \diamond)$. On the other hand, if $\delta(q, i)$ is defined, for some i, $1 \leq i \leq k$, or $\delta(q, \varepsilon)$ is defined, then we do not add any transition. By definition, M' is complete and, since t is non-accepting, $\mathcal{L}(M) = \mathcal{L}(M')$. ◀

▶ Remark. We note that the construction of the proof of Proposition 37 preserves completeness, i. e., if M is a complete DTMFA, then we obtain an equivalent complete DTMFA without ε-transitions. Moreover, the construction of the proof of Proposition 38 does not introduce ε-transitions; thus, it turns an ε-free DTMFA into an equivalent complete DTMFA that is still ε-free.

We are now ready to give the proof of Theorem 6.

Proof. Let $M = (Q, \Sigma, \delta, q_0, F) \in \mathsf{DTMFA}$. By Proposition 38, we can assume that M is complete. Due to Proposition 36, for any input w, there is a unique computation of M on w. Hence, the idea is now to toggle the acceptance of all the states of M in order to obtain a DTMFA that accepts $\overline{\mathcal{L}(M)}$. However, this only works if M is ε-free, since otherwise it is possible that some word $w \in \Sigma^*$ cannot be fully consumed by M (for example, if it leads into a loop in which all transitions are ε-transitions and no state is accepting); thus, w is neither accepted by M nor by the DTMFA obtained by toggling the acceptance of states. While we can remove ε-transitions due to Proposition 37, we encounter the problem that a memory recall transition with respect to an empty memory behaves just like an ε-transition and, thus, can cause the same problems. Hence, we first have to transform such memory recall transition into ordinary ε-transitions, which can then be removed according to Proposition 37.

To this end, we modify M such that the finite state control stores, for every i, $1 \leq i \leq k$, whether or not memory i is open and whether or not memory i stores the empty word. More precisely, we obtain an $M_1 \in \mathsf{DTMFA}$ by modifying M as follows. Every state q is replaced by 2^{2k} new states $[q, (r_1, c_1), \ldots, (r_k, c_k)]$, where $r_i \in \{\mathtt{C}, \mathtt{O}\}$, $c_i \in \{\varepsilon, \overline{\varepsilon}\}$, $1 \leq i \leq k$, and we change the transitions such that if M_1 reaches a configuration with state $[q, (r_1, c_1), \ldots, (r_k, c_k)]$, then, in the current configuration, for every i, $1 \leq i \leq k$, r_i is the status of memory i and memory i is empty if and only if $c_i = \varepsilon$. For example, if M_1 is in state $[p, (r_1, c_1), \ldots, (r_k, c_k)]$ with $(r_i, c_i) = (\mathtt{C}, \overline{\varepsilon})$ and $\delta(p, x) = (q, s_1, \ldots, s_k)$ with $x \in \Sigma$ and $s_i = \mathtt{o}$, then, if x is the next symbol of the input, M_1 changes to a state $[q, (r'_1, c'_1), \ldots, (r'_k, c'_k)]$ with $(r'_i, c'_i) = (\mathtt{O}, \overline{\varepsilon})$. We note that M_1 is still complete and deterministic.

Next, we change M_1 into M_2 by replacing, for every i, $1 \leq i \leq k$, every transition of the form $\delta([p, (r_1, c_1), \ldots, (r_k, c_k)], i) = ([q, (r'_1, c'_1), \ldots, (r'_k, c'_k)], s_1, \ldots, s_k)$ with $c_i = \varepsilon$ by an ε-transition $\delta([p, (r_1, c_1), \ldots, (r_k, c_k)], \varepsilon) = ([q, (r'_1, c'_1), \ldots, (r'_k, c'_k)], s_1, \ldots, s_k)$. We note that $\mathcal{L}(M_1) = \mathcal{L}(M_2)$ and, since M_1 is deterministic, this only introduces ε-transitions, such that if $\delta(p, \varepsilon)$ is defined then, for every $y \in (\Sigma \cup \{1, 2, \ldots, k\})$, $\delta(q, y)$ is undefined. Consequently, M_2 is still deterministic and it never happens that an empty memory is recalled. Next, by Proposition 37, we can transform M_2 into a complete $M_3 \in \mathsf{DTMFA}$ without ε-transitions (see Remark A.4) that still has the property that no empty memories are recalled.

Let $\overline{M} \in \mathsf{DTMFA}$ be obtained from M_3 by toggling the acceptance of the states, i. e., if Q_3 and F_3 are the sets of states and accepting states, respectively, of M_3, then $\overline{M}$ is obtained from M_3 by replacing F_3 by $Q_3 \setminus F_3$. Obviously, for every $w \in \Sigma^*$, both M_3 and $\overline{M}$, on input w, reach the same state and completely consume the input. This directly implies $\mathcal{L}(\overline{M}) = \overline{\mathcal{L}(M_3)}$. ◀

A.5 Proof of Theorem 5

Proof. We first modify M with respect to its ε-transitions as follows. Let $p \in Q$ be a state with an ε-transition that is followed by another ε-transition. If p is contained in a cycle $q_1, q_2, \ldots, q_n$ of ε-transitions, we simply replace this cycle by a single state q' (i. e., all incoming edges of any q_i, $1 \leq i \leq n$, then point to q') that is accepting if and only if some q_i, $1 \leq i \leq n$, is (note that, since M is deterministic, no q_i has any other transition). Otherwise, there are states $q_1, q_2, \ldots, q_n$, $p = q_1$, with transitions $\delta(q_i, \varepsilon) = (q_{i+1}, s_{i,1}, \ldots, s_{i,k})$, $1 \leq i \leq n-1$, such that q_n has no ε-transition. We can now remove the transition $\delta(q_1, \varepsilon) = (q_2, s_{1,1}, \ldots, s_{1,k})$ and add a transition $\delta(q_1, \varepsilon) = (q_n, t_1, \ldots, t_k)$, where, for every j, $1 \leq j \leq k$, t_j is a memory instruction that has the same effect as applying instructions $s_{1,j}, s_{2,j}, \ldots, s_{n-1,j}$ in this order. Moreover, if, for some i, $2 \leq i \leq n-1$, $q_i \in F$, then we define q_1 as accepting. By applying this modification for every ε-transition that is followed by another ε-transition, we can modify M such that no ε-transition is followed by another ε-transition. Hence, since M is deterministic, there are at most $|Q|$ ε-transitions and for each, we have to determine the states $q_1, q_2, \ldots, q_n$ and perform the modifications described above, which can be done in time $O(|Q|)$, as well. Consequently, the whole procedure can be carried out in $O(|Q|^2)$.

Next, we consider states with a memory recall transition. Similar as for states with ε-transition, such states are followed by a (possibly empty) sequence of consecutive memory recall or ε-transitions that either ends in a state with neither memory recall nor ε-transition or eventually forms a loop. We first consider the case, where this sequence does not contain any ε-transitions and does not form a loop. Let q_1 be the state with memory recall transition and let $(q_1, \ell_1), (q_2, \ell_2), \ldots, (q_n, \ell_n), q_{n+1}$ be the sequence of the following states with consecutive memory recall transitions along with the memory that is recalled. More precisely, the transition from q_i to q_{i+1}, $1 \leq i \leq n$, recalls ℓ_i and the last element q_{n+1} is the first state without memory recall transition (and, by assumption, also without ε-transition). We now contract this list by the following algorithm. Initially, let $A = \emptyset$. Then we move through the list from left to right and for every element (q_i, ℓ_i) (except for q_{n+1}), we proceed as follows. If $\ell_i \in A$, then we remove (q_i, ℓ_i) and if $\ell_i \notin A$, then we keep (q_i, ℓ_i) and add ℓ_i to A. Obviously, this results in a list $(p_1, r_1), \ldots, (p_{n'}, r_{n'}), q_{n+1}$ with $n' \leq k$. The idea is that if we move from left to right through this new list, it tells us which state to enter if the memory of the current memory recall is empty, i. e., if memory r_1 is non-empty, we recall it in state p_1, if memory r_1 is empty, we can directly jump to state p_2 and either recall r_2, if it is non-empty, or jump to p_3 otherwise, and so on. If all memories (that occur somewhere in the list) are empty, we end up in state q_{n+1}.

In the presence of ε-transitions, we simply ignore these and always only consider the next transition that recalls a memory, i. e., it is possible that for elements (q_i, ℓ_i) and (q_{i+1}, ℓ_{i+1}) of the non-contracted list, there is an intermediate state p with recall transition from q_i to p and ε-transitions from p to q_{i+1} (note that due to the construction from above, there are no consecutive ε-transitions), but the contraction works in the same way. Moreover, if (q_i, ℓ_i) and (q_j, ℓ_j) (or q_{n+1}, the last element) with $i < j$ are consecutive elements of the contracted list (i. e., all elements (q_r, ℓ_r), $i + 1 \leq r \leq j - 1$, have been deleted by the algorithm), then we replace (q_i, ℓ_i) by $(q_i^{\text{ACC}}, \ell_i)$ (where the marker ACC means that we can accept), if for some r, $i + 1 \leq r \leq j$, $(q_r, \ell_r) \in F$. Note that this is analogous to the modification from above, where we define states as accepting, if they are connected to an accepting state by a sequence of ε-transitions, but here we cannot change acceptance of the actual states, since it depends on the current contents of memories, whether we can reach an accepting state by only recalls of empty memories or ε-transitions.

If the sequence of memory recall transitions enters a loop, we construct the list only up to the first time a state is repeated, say p, and have (p, LOOP) as the last element of the list. Then we apply the contraction in the same way as before, where (p, LOOP) plays the role of q_{n+1}. Similarly as before, we mark elements (q_i, ℓ_i) as accepting if a pair was removed that contained an accepting state.

In addition to the states, we also store in the list the memory instructions that have to be applied in order to jump to the next state (this can be done similiar as for contracting the ε-transitions above). We construct such a list for every state with a memory recall transition. Every single list can be constructed in time $O(|Q|)$, so we need time $O(|Q|^2)$ in total.

Now we check whether or not $w \in \mathcal{L}(M)$ by running (the modified) M on input w in a special way. We first initialise a list $(1, \mathtt{C}, \varepsilon), (2, \mathtt{C}, \varepsilon), \ldots, (k, \mathtt{C}, \varepsilon)$ indicating that every memory is closed and empty. Then we simulate M on input w as follows. Every transition that consumes a single symbol as well as every ε-transition is just carried out. Whenever a memory status is changed, we store this in the list and we also store whether a memory is currently empty or not (note that we have to know the current statuses in order to do this). When a memory is recalled in state q, then we move through the list stored for state q until we find a recall of a memory that is currently non-empty, jump in the automaton to the corresponding state and apply the memory instructions. Whenever we reach an element $(q_i^{\text{ACC}}, \ell_i)$ in the list, then we check whether the input has been fully consumed and if yes, we conclude $w \in \mathcal{L}(M)$. If we reach in a list an element (p, LOOP), then we conclude $w \in \mathcal{L}(M)$, if $p \in F$ and $w \notin \mathcal{L}(M)$ otherwise. If in the computation the input has been completely consumed, then we conclude $w \in \mathcal{L}(M)$ if and only if M is in an accepting state.

Since there are no consecutive ε-transitions, every consumption of a single symbol from the input by a transition is done in constant time. Every consumption by a memory recall transition requires time $O(k)$, since we have to move through a list of size $O(k)$. Consequently, the total running time is $O(|Q|^2 + k|w|)$. ◀

A.6 Proof of Lemma 7

Proof. As $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$, there exists an $M \in \mathsf{DTMFA}^{\mathsf{rej}}$ with $\mathcal{L}(M) = L$. If there is an $m \geq 0$ such that in every accepting run of M, each variable stores only a word of length at most m, then L is regular (as we can rewrite M into a DFA that stores the contents of the variables in its states). Likewise, if variables can store words of unbounded length, but are then never recalled, these variables can be eliminated, which also allows us to turn M into a DFA for L.

Hence, if L is not regular, M has at least one variable x such that for every $m \geq 0$, there is an accepting run of M on a word w during which x stores a word of length $n \geq m$, and this variable is recalled with this content. Let p_n be the part of the accepting run that M has processed up to a state q where it recalls x at a point where this variable contains a word of length n. Let v_n be this content of x (hence, $|v_n| = n$). As v_n must have been consumed while processing p_n, $|p_n| \geq n$ holds, and v_n must be a factor of p_n.

If M succeeds at recalling x at this point (i. e., it consumes v_n), it can continue to accept w, which means that $p_n v_n$ is a prefix of $w \in L$. On the other hand, for each $u \in \Sigma^+$ such that v_n is not a prefix of u, M encounters a memory recall failure and rejects. As M is deterministic, the recall transition for x must be the only transition that leaves the state q. Hence, $p_n u \notin L$ for u that do not have v_n as prefix. ◀

A.7 Proof of Lemma 10

Proof. We show that 1 implies 2, which implies 3, which implies 1. The first two of these steps are simple: Assume that L is regular. Every regular language over a single letter alphabet can be expressed as a finite union of arithmetic progressions (cf., e. g., Chrobak [13, 14]). As L is infinite, it must contain an infinite arithmetic progression. But if L contains an infinite arithmetic progression $\mathtt{a}^{ib+c}$, then the third condition is satisfied by definition.

The step from 3 to 1 is more involved. Before we prove this, note that there are unary languages (which are not DTMFA-languages), for which condition 3 does not imply the existence of an infinite arithmetic progression, see Example 39 below.

Assume that $L \subseteq \{\mathtt{a}\}^*$ is infinite, $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$, and condition 3 is met is for some $b \geq 1$. By definition, there is an $M \in \mathsf{DTMFA}^{\mathsf{rej}}$ with $\mathcal{L}(M) = L$. As M is deterministic, each of its states can have at most one outgoing transition; and as L is infinite, each state must have exactly one outgoing transition. Hence, like a DFA for a unary language (see e. g. the proof of Theorem 25), M consists of a chain and a cycle. Let m be the number of accepting states on the cycle, and let k be the number of variables that are accessed in the cycle (by recalling them, or by performing memory instructions).

Now consider an $n > m(b+1)^k$ such that there exists a c_n with $w_i := \mathtt{a}^{bi+c_n} \in L$ for all $0 \leq i \leq n$, and reading $w_0 = \mathtt{a}^{b+c_n}$ takes M into the cycle (as condition 3 holds for all n, such a c_n exists for every n that is sufficiently large). Let q be the accepting state that is reached by w_0.

In the following, by an *iteration* of the cycle, we mean the situation that M is in state q and then consumes input symbols until it reaches q for the next time. The iteration of the cycle that starts after having fully consumed w_0 is called iteration 1. Now, for every $j \geq 1$, we define a function $\vec{v}_j : \{1, \ldots, k\} \to \mathbb{N}$ that describes the content of each memory after completing iteration j.

In the remainder of the proof, we show that there is a constant upper bound for the values $\vec{v}_j(x)$, $1 \leq x \leq k$, $j \geq 1$. Note that if the length of the content of each memory is bounded, then M can be rewritten into an equivalent DFA that simulates all memories in its states. Hence, L must be a regular language, which shows that condition 3 implies condition 1.

As M has to accept all words w_i with $0 \leq i \leq n$, and as each iteration of the cycle can accept only m words, we know that M has to perform at least $I := \frac{n}{m} > (b+1)^k$ iterations of the cycle in order to accept w_n. During these iterations, M cannot consume more than $\mathtt{a}^b$ between each pair of accepting states – otherwise, M would skip at least one of the w_i (as M is deterministic, the run for w_n must be an extension of each run for a w_i with $i < n$). In particular, this means that each memory that is recalled during these iterations cannot contain more than $\mathtt{a}^b$; thus, there are only $b+1$ possible contents for each memory. Furthermore, as M is deterministic, we know that each memory that is not recalled during these iterations will not be recalled during any later iterations of the cycle, which means that it can be removed from the cycle (and, as the chain is of finite length, it can also be removed from the chain). Hence, without loss of generality, we can assume that $\vec{v}_j(x) \leq b$, $1 \leq x \leq k'$, $1 \leq j \leq I$, where the cycle contains exactly the memories $1, \ldots, k$.

As $I > (b+1)^k$ and as there are only $(b+1)^k$ possible choices of $\vec{v}_j$, there exist j, j' with $0 \leq j' < j' \leq I$ and $\vec{v}_j = \vec{v}_{j'}$. As M is deterministic, this allows us to conclude $\vec{v}_{j+l} = \vec{v}_{j'+l}$ for all $l \geq 0$. In other words, the sequence of transitions from iteration j to iteration j' will be repeated forever, using exactly the same memory contents, which means that $\vec{v}_l(x) \leq b$ for all $l \geq 0$ and all $1 \leq x \leq k$. As explained above, this concludes the proof. ◀

▶ **Example 39.** We define a $L_{\mathsf{ex}} \subset \{\mathtt{a}\}^*$ together with its complement $\overline{L_{\mathsf{ex}}}$ in the following way: First, add $\mathtt{a}$ to L_{ex}, then add the two words $\mathtt{a}^2$ and $\mathtt{a}^3$ to $\overline{L_{\mathsf{ex}}}$, and the three words $\mathtt{a}^4$ to $\mathtt{a}^6$ to L_{ex}, and so on. In other words, in each step i, we add the next i words of $\{\mathtt{a}\}^*$ to one of the languages; namely L_{ex} if i is odd, and $\overline{L_{\mathsf{ex}}}$ if i is even. Then L_{ex} satisfies condition 3 of Lemma 10, but it does not contain any infinite arithmetic progression. Hence, $L_{\mathsf{ex}} \notin \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$; and $\overline{L_{\mathsf{ex}}} \notin \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ follows analogously.

A.8 Proof of Proposition 12

Proof. We first observe that $\mathcal{L}(\mathsf{REG}) \subseteq \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \cap \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$ holds by definition. Next, we assume that there is a non-regular $L \in (\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \cap \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}}))$ over $\{\mathtt{a}\}^*$. In particular, this implies that both L and its complement $\overline{L} := \{\mathtt{a}\}^* \setminus L$ are infinite and, furthermore, by Theorem 6, $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$ implies $\overline{L} \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$. Since $\overline{L} \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ is a non-regular $\mathsf{DTMFA}^{\mathsf{rej}}$ language, Lemma 7 allows us to conclude that for every $m \geq 0$, there exist an $n \geq m$ and a $p_n \geq n$ such that $\mathtt{a}^i \notin L$ for all $p_n \leq i < p_n + n$.

Hence, $\overline{L}$ contains finite arithmetic progressions of unbounded length; and as $\overline{L}$ is infinite, Lemma 10 states that $\overline{L}$ is regular, which is a contradiction. ◀

A.9 Proof of Theorem 17

Proof. We construct $\mathcal{M}(\alpha)$ by first constructing a graph $G_{\tilde{\alpha}}$ from the marked regex $\tilde{\alpha}$. As $G_{\tilde{\alpha}}$ is a generalization of the occurrence graphs for proper regular expressions, we call this the *memory occurrence graph*. Analogously to proper regular expressions, this graph can be directly interpreted as an $\mathcal{M}(\alpha) \in \mathsf{TMFA}^{\mathsf{rej}}$ that is deterministic if and only if α is deterministic.

Memory occurrence graph $G_{\tilde{\alpha}}$: Given a marked regex $\tilde{\alpha}$, we define a memory occurrence graph $G_{\tilde{\alpha}} := (V_{\tilde{\alpha}}, E_{\tilde{\alpha}})$ with a source node src, a sink node snk, and one node for each marked variable reference or terminal. The labeled edges are of the form (u, ν, v), where $u, v \in V_{\tilde{\alpha}}$, and each label ν is a marked ref-word $\nu \in \tilde{\Gamma}^*$. We use marked ref-words instead of unmarked ref-words to fulfill the promise that $\mathcal{M}(\alpha)$ is deterministic if and only if α is deterministic. If α has n occurrences of variable references and terminals, $\mathcal{M}(\alpha)$ has $n+2$ states: the initial state, the state $[\mathsf{trap}]$ for memory recall failures, and one state for each of the n occurrences in α.

If we only want to construct an algorithm that turns a deterministic regex into a $\mathsf{DTMFA}^{\mathsf{rej}}$ and rejects non-deterministic regexes, we can use unmarked edge labels instead (see the section at the end of this proof).

When interpreting $G_{\tilde{\alpha}}$ as a TMFA $\mathcal{M}(\alpha)$, we first remove the markings from the edge labels, and interpret these as memory actions of a TMFA, e. g., $[_x$ corresponds to opening the memory for x. In order to simplify the construction, we take into account that different ref-words over Γ can have the same net effect on variables, and can be represented by the same single transition in a TMFA. For example, $]_x[_x]_x[_x$ and $]_x[_x$ and $[_x$ all have the same effect as performing $\mathtt{o}$ on the memory for x. Following this intuition, given a ref-word $\nu \in \Gamma^*$, we define the *net variable action of* ν as a function $\mathsf{net}_\nu \colon \Xi \to \{\mathtt{o}, \mathtt{c}, \mathtt{r}, \diamond\}$, where for each $x \in \Xi$, $\mathsf{net}_\nu(x) := \diamond$ if no element of $\Gamma_x := \{[_x,]_x\}$ occurs in ν, and $\mathsf{net}_\nu(x) := \mathtt{o}$ if the rightmost occurrence of an element of Γ_x is a $[_x$. Furthermore, if the rightmost occurrence of an element of Γ_x is $]_x$, we define $\mathsf{net}_\nu(x) := \mathtt{r}$ if ν contains $[_x$, and $\mathsf{net}_\nu(x) := \mathtt{c}$ otherwise. In the construction further down, we also consider concatenations of labels. We observe the following for all $\nu, \nu_1, \nu_2 \in \Gamma^*$ and all $x \in \Xi$: If $\mathsf{net}_\nu(x) = \diamond$, then $\mathsf{net}_{\nu_1 \cdot \nu}(x) = \mathsf{net}_{\nu_1}(x)$ and $\mathsf{net}_{\nu \cdot \nu_2} = \mathsf{net}_{\nu_2}(x)$. If $\mathsf{net}_{\nu_2}(x) \in \{\mathtt{o}, \mathtt{r}\}$, then $\mathsf{net}_{\nu_1 \cdot \nu_2}(x) = \mathsf{net}_{\nu_2}(x)$.

We also use the following notion of minimal representations: For all $\nu \in \Gamma^*$ and $x \in \Xi$, we define $\mathsf{min}_x(\nu) \in \Gamma^*$ by $\mathsf{min}_x(\nu) := [_x$ if $\mathsf{net}_\nu(x) = \mathtt{o}$, $\mathsf{min}_x(\nu) :=]_x$ if $\mathsf{net}_\nu(x) = \mathtt{c}$, $\mathsf{min}_x(\nu) := [_x]_x$ if $\mathsf{net}_\nu(x) = \mathtt{r}$, and $\mathsf{min}_x(\nu) := \varepsilon$ if $\mathsf{net}_\nu(x) = \diamond$. For any $\nu \in \Gamma^*$, its minimal representation $\mathsf{min}(\nu)$ is defined as any concatenation of all $\mathsf{min}_x(\nu)$ for all $x \in \Xi$ (as $\mathsf{net}_\nu \neq \diamond$ holds only for finitely many $x \in \Xi$, this is not problematic). In other words, for each $\nu \in \Gamma^*$, $\mathsf{min}(\nu)$ is one of the shortest words in Γ^* that satisfies $\mathsf{net}_{\mathsf{min}(\nu)} = \mathsf{net}_\nu$.

By using net, we can directly interpret a memory occurrence graph $G_{\tilde{\alpha}}$ as a TMFA $\mathcal{M}(\alpha) := (Q, \Sigma, \delta, \mathsf{src}, F)$, analogously to the occurrence graph for proper regular expressions. The components of $\mathcal{M}(\alpha)$ are obtained as follows: First, we rename the variables such that $G_{\tilde{\alpha}}$ contains exactly the variables $\{1, \ldots, k\}$ for some $k \geq 0$ (hence, for each $1 \leq i \leq k$, there is a variable $x_i \in \mathsf{var}(\alpha)$ such that x_i is represented by 1). We then define

$$Q := (V_{\tilde{\alpha}} \setminus \{\mathsf{snk}\}) \cup \{[\mathsf{trap}]\},$$

$$F := \{u \in Q \mid (u, \nu, \mathsf{snk}) \in E_{\tilde{\alpha}} \text{ for some } \nu\}.$$

In other words, all nodes except snk are states, and all nodes that have an edge to snk are final states (as in the occurrence graph). Following this intuition, each edge (u, ν, v) with $v \neq \mathsf{snk}$ corresponds to a transition from state u to state v, while performing the memory actions of $\mathsf{net}_\nu(x)$ on each $x \in \mathsf{var}(\alpha)$. In order to allow recursive applications of the construction, each edge (u, ν, snk) not only marks that u is an accepting state, but also that the memory actions of ν need to be performed before accepting. Formally, we define δ to include exactly the following transitions:

1. If $(u, \nu, a_{(i)}) \in E_{\tilde{\alpha}}$ with $a \in \Sigma$, then $(a_{(i)}, s_1, \dots, s_k) \in \delta(u, a)$.
2. If $(u, \nu, x_{(i)}) \in E_{\tilde{\alpha}}$ with $x \in \mathsf{var}(\alpha)$, then $(x_{(i)}, s_1, \dots, s_k) \in \delta(u, x)$,

where for each $1 \leq i \leq k$, $s_i := \mathsf{net}_{\mathsf{unmark}(\nu)}(x_i)$ (unless the transition recalls memory i; then we choose $s_i := \mathsf{c}$ as required by Definition 3). Recall that $\mathsf{unmark}\colon (\tilde{\Sigma} \cup \tilde{\Xi} \cup \tilde{\Gamma}) \to (\Sigma \cup \Xi \cup \Gamma)^*$ is the morphism that removes the markings from marked letters. As we shall see, in order to satisfy the condition that $\mathcal{M}(\alpha)$ is deterministic *only if* α is deterministic, we need to slightly adapt this definition.

Following this interpretation, we say that a memory occurrence graph $G_{\tilde{\alpha}}$ is *not deterministic* if there exists a $u \in V_{\tilde{\alpha}}$ such that any of the following conditions is met:

1. $E_{\tilde{\alpha}}$ contains edges $(u, \nu_1, a_{(i)})$ and $(u, \nu_2, a_{(j)})$ with $i \neq j$ and $a \in \Sigma$.
2. $E_{\tilde{\alpha}}$ contains edges $(u, \nu_1, x_{(i)})$ and $(u, \nu_2, \chi_{(j)})$ with $i \neq j$, $x \in \Xi$, $\chi \in (\Xi \cup \Sigma)$,
3. $E_{\tilde{\alpha}}$ contains edges $(u, \nu_1, \chi_{(i)})$ and $(u, \nu_2, \chi_{(i)})$ with $\nu_1 \neq \nu_2$ and $\chi \in (\Xi \cup \Sigma)$,
4. $E_{\tilde{\alpha}}$ contains edges (u, ν_1, snk) and (u, ν_2, snk) with $\nu_1 \neq \nu_2$.

Otherwise, we call $G_{\tilde{\alpha}}$ *deterministic*. It is easily seen that if $G_{\tilde{\alpha}}$ is deterministic, $\mathcal{M}(\alpha)$ is also deterministic. For the other direction, we need to account for two problems: First, it is possible that two labeled ref-words ν_1 and ν_2 map to the same memory action $\mathsf{net}_{\mathsf{unmark}(\nu_1)} = \mathsf{net}_{\mathsf{unmark}(\nu_2)}$, e. g., $\nu_1 = [_{x1}]_{x(2)}$ and $\nu_2 = [_{x3}]_{x(4)}[_{x5}]_{x(6)}$, which can occur in regex like $\alpha_1 := \big(\langle x\colon \varepsilon\rangle \vee (\langle x\colon \varepsilon\rangle\langle x\colon \varepsilon\rangle)\big)\mathsf{a}$. Second, as $\mathcal{M}(\alpha)$ has no ε-transitions, it does not model the difference between distinct edges to snk, as they appear when converting regex like $\alpha_2 := (\varepsilon \vee \langle x\colon \varepsilon\rangle)$. As DTMFA cannot detect explicitly that the end of the input has been reached, they cannot simulate the memory actions of edges to snk, which means that the construction ignores this.

In both cases, the accepted language is correct; but this has the side effect the resulting $\mathcal{M}(\alpha)$ is deterministic, although α and $G_{\tilde{\alpha}}$ are not. Hence, to ensure that $\mathcal{M}(\alpha)$ is deterministic only if α is deterministic, we proceed as follows: If $G_{\tilde{\alpha}}$ contains any of these edges, we pick any transition $\delta(q, b) \ni (p, s_1, \dots, s_k)$ with $b \in \Sigma \cup \{1, 2, \dots, k\}$. We then add a new state p_{ndet}, a transition $\delta(q, b) \ni (p_{\mathsf{ndet}}, s_1, \dots, s_k)$, and p_{ndet} has the same outgoing transitions as p. If we want to construct an algorithm that rejects non-deterministic regex, we can simply omit this technical crutch, and detect these cases in the construction of $G_{\tilde{\alpha}}$ as discussed below.

Constructing $G_{\tilde{\alpha}}$: We now define $G_{\tilde{\alpha}} = (V_{\tilde{\alpha}}, E_{\tilde{\alpha}})$ recursively.

1. **Empty word:** If $\tilde{\alpha} = \varepsilon$, we define

$$V_{\tilde{\alpha}} := \{\mathsf{src}, \mathsf{snk}\},$$
$$E_{\tilde{\alpha}} := \{(\mathsf{src}, \varepsilon, \mathsf{snk})\}.$$

This case is completely straightforward: An edge from src to snk is how occurrence graphs model ε, and the marking ε means that this transition performs no memory actions.

2. **Terminals and variable references:** If $\tilde{\alpha} = \chi_{(i)}$ with $\chi \in (\Sigma \cup \Xi)$, we define

$$V_{\tilde{\alpha}} := \{\mathsf{src}, \chi_{(i)}, \mathsf{snk}\},$$
$$E_{\tilde{\alpha}} := \{(\mathsf{src}, \varepsilon, \chi_{(i)}), (\chi_{(i)}, \varepsilon, \mathsf{snk})\}.$$

Similar to the case for ε, this models that the terminal is read, or that a variable reference is processed, by recalling the appropriate memory.

3. **Variable bindings:** If $\tilde{\alpha} = ([_{x(i)}\tilde{\beta}]_{x(j)})$ with $x \in \Xi$, we define $V_{\tilde{\alpha}} := V_{\tilde{\beta}}$ and

$$\begin{aligned}E_{\tilde{\alpha}} := &\{(\mathsf{src}, [_{x(i)} \cdot \nu_{\mathsf{in}}, v) \mid (\mathsf{src}, \nu_{\mathsf{in}}, v) \in E_{\tilde{\beta}}, v \neq \mathsf{snk}\}\\ &\cup \{(u, \nu, v) \mid (u, \nu, v) \in E_{\tilde{\beta}}, u \neq \mathsf{src}, v \neq \mathsf{snk}\}\\ &\cup \{(u, \nu_{\mathsf{out}} \cdot]_{x(j)}, \mathsf{snk}) \mid (u, \nu_{\mathsf{out}}, \mathsf{snk}) \in E_{\tilde{\beta}}, u \neq \mathsf{src}\}\\ &\cup \{(\mathsf{src}, [_{x(i)} \cdot \nu_{\varepsilon} \cdot]_{x(j)}, \mathsf{snk}) \mid (\mathsf{src}, \nu_{\varepsilon}, \mathsf{snk}) \in E_{\tilde{\beta}}\},\end{aligned}$$

Less formally, we take the memory occurrence graph for β and add opening (and closing) of x to all edges from src (and to snk, respectively); while all other edges remain unchanged. Note that for edges from src to snk, we could also use $\nu_{\varepsilon} \cdot [_x]_x$ or $[_x]_x \cdot \nu_{\varepsilon}$, as by Definition 1, $\langle x\colon \beta\rangle$ is only a regex if $x \notin \mathsf{var}(\beta)$, which implies that no marked $[_x$ or $]_x$ occurs in ν_{ε}.

4. **Disjunction:** If $\tilde{\alpha} = (\tilde{\beta} \vee \tilde{\gamma})$, we define $V_{\tilde{\alpha}} := V_{\tilde{\beta}} \cup V_{\tilde{\gamma}}$ and $E_{\tilde{\alpha}} := E_{\tilde{\beta}} \cup E_{\tilde{\gamma}}$.
As the markings define a one to one correspondence between the nodes in $V_{\tilde{\alpha}}$ and the terminals and the variable references in α, we know that $V_{\tilde{\beta}} \cap V_{\tilde{\gamma}} = \{\mathsf{src}, \mathsf{snk}\}$. Therefore, the resulting memory occurrence graph $G_{\tilde{\alpha}}$ computes the union of $G_{\tilde{\beta}}$ and $G_{\tilde{\gamma}}$.

5. **Concatenation:** If $\tilde{\alpha} = (\tilde{\beta} \cdot \tilde{\gamma})$, we define

$$\begin{aligned}V_{\tilde{\alpha}} :=& V_{\tilde{\beta}} \cup V_{\tilde{\gamma}},\\ E_{\tilde{\alpha}} :=&\{(u, \nu, v) \mid (u, \nu, v) \in E_{\tilde{\beta}}, v \neq \mathsf{snk}\}\\ &\cup \{(u, \nu, v) \mid (u, \nu, v) \in E_{\tilde{\gamma}}, u \neq \mathsf{src}\}\\ &\cup \{(u, (\nu_1 \cdot \nu_2), v) \mid (u, \nu_1, \mathsf{snk}) \in E_{\tilde{\beta}}, (\mathsf{src}, \nu_2, v) \in E_{\tilde{\gamma}}\},\end{aligned}$$

Again, we use the fact that $V_{\tilde{\beta}} \cap V_{\tilde{\gamma}} = \{\mathsf{src}, \mathsf{snk}\}$. The memory occurrence graph $G_{\tilde{\alpha}}$ first simulates $G_{\tilde{\beta}}$, until the latter would accept by processing an edge $(u, \nu_1, \mathsf{snk}) \in E_{\tilde{\beta}}$. Instead of following this edge to snk, $G_{\tilde{\alpha}}$ then starts its simulation of $G_{\tilde{\gamma}}$, by picking any edge $(\mathsf{src}, \nu_2, v) \in E_{\tilde{\gamma}}$, which is merged with (u, ν_1, snk) into a single edge from u to v, and its label is $(\nu_1 \cdot \nu_2)$. Hence, it is easy to see that $G_{\tilde{\alpha}}$ computes the concatenation of $G_{\tilde{\beta}}$ and $G_{\tilde{\gamma}}$.

6. **Kleene plus:** Assume $\tilde{\alpha} = \tilde{\beta}^+$. This case requires some additional definitions. Let N_{ε} denote the set of all ν with $(\mathsf{src}, \nu, \mathsf{snk}) \in E_{\tilde{\beta}}$, and let $N^{(*)} := \{\mathsf{min}(\nu) \mid \nu \in N_{\varepsilon}^*\}$, where we assume that the elements of $N^{(*)}$ have some arbitrary markings (as we shall see, this definition matters only for non-deterministic regex, which means that we do not need markings to detect non-determinism).
Note that, as N_{ε} is finite, there are only finitely many $x \in \Xi$ such that $\mathsf{net}_{\nu}(x) \neq \diamond$ for a $\nu \in N_{\varepsilon}$, which implies that $N^{(*)}$ is finite. In order to avoid hiding non-determinism in some very special cases, we assume that $N^{(*)}$ always contains at least two elements (this is possible without loss of generality, as we can always add some ν^2 for a $\nu \in N^{(*)}$ without changing the behavior). We now define $V_{\tilde{\alpha}} := V_{\tilde{\beta}}$, as well as

$$\begin{aligned}E_{\tilde{\alpha}} := E_{\tilde{\beta}} &\cup \{(\mathsf{src}, \hat{\nu} \cdot \nu_{\mathsf{in}}, v) \mid (\mathsf{src}, \nu_{\mathsf{in}}, v) \in E_{\tilde{\beta}}, \hat{\nu} \in N^{(*)}\}\\ &\cup \{(u, \nu_{\mathsf{out}} \cdot \hat{\nu}, \mathsf{snk}) \mid (u, \nu_{\mathsf{out}}, \mathsf{snk}) \in E_{\tilde{\beta}}, \hat{\nu} \in N^{(*)}\}\\ &\cup \{(u, \nu_{\mathsf{out}} \cdot \hat{\nu} \cdot \nu_{\mathsf{in}}, v) \mid (u, \nu_{\mathsf{out}}, \mathsf{snk}) \in E_{\tilde{\beta}}, (\mathsf{src}, \nu_{\mathsf{in}}, v) \in E_{\tilde{\beta}}, \hat{\nu} \in N^{(*)}\}.\end{aligned}$$

Similar to the construction for concatenation, the idea is that $G_{\tilde{\alpha}}$ simulates $G_{\tilde{\beta}}$; and whenever the latter could accept by taking an edge to snk, the former can loop back to the beginning. The only difficult part is when $G_{\tilde{\beta}}$ contains edges from src to snk with memory actions. As the Kleene plus allows us to use an arbitrary amount of these edges before taking an edge from src or to snk, we need to include N^*_ε in the functions. This set is generally infinite; but it can be compacted to the finite set $N^{(*)}$.
For deterministic regex, this construction collapses to a far simpler case that does not use $N^{(*)}$: First, note that if α is deterministic and contains β^+, $(\mathsf{src}, \nu, \mathsf{snk}) \in E_{\tilde{\beta}}$ implies $\nu = \varepsilon$, as otherwise, β would satisfy condition 4 of Definition 14, and α would satisfy condition 3 or 4. Hence, if α is deterministic, we can assume that or $N_\varepsilon = \{\varepsilon\}$ or $N_\varepsilon = \emptyset$; both cases lead to $N^{(*)} = \{\varepsilon\}$. This allows us to use the following simplified definition:

$$E_{\tilde{\alpha}} := E_{\tilde{\beta}} \cup \{(u, \nu_{\mathsf{out}} \cdot \nu_{\mathsf{in}}, v) \mid (u, \nu_{\mathsf{out}}, \mathsf{snk}) \in E_{\tilde{\beta}}, (\mathsf{src}, \nu_{\mathsf{in}}, v) \in E_{\tilde{\beta}}\}.$$

Hence, when constructing $G_{\tilde{\alpha}}$ inductively, we first check if $E_{\tilde{\beta}}$ contains an edge (u, ν, v) with $\nu \neq \varepsilon$. If this is the case, we can reject α as not deterministic. Otherwise, we use this simplified definition.

Correctness and determinism: The correctness of the construction is easily seen by a lengthy but straightforward induction, using the explanations provided with the definitions above. In particular, note that if α does not contain a Kleene plus (or contains a Kleene plus and is deterministic), each path from src to snk through $G_{\tilde{\alpha}}$ corresponds to a marked ref-word from $\mathcal{R}(\alpha)$, and vice versa. If α is not deterministic and contains a Kleene plus, the correspondence is a little bit less strict, as ref-words $\gamma \in \Gamma^+$ are compressed to the equivalent $\mathsf{min}(\gamma)$.

To see that $\mathcal{M}(\alpha)$ is deterministic if and only if α is deterministic, recall that we established above that $\mathcal{M}(\alpha)$ is deterministic if and only if $G_{\tilde{\alpha}}$ is deterministic. Hence, it suffices to show that if determinism in $G_{\tilde{\alpha}}$ is equivalent to determinism in α. But this follows immediately from our observation that there is a one-to-one correspondence between paths in $G_{\tilde{\alpha}}$ and the marked ref-words in $\mathcal{R}(\alpha)$, and the fact that each node $\chi_{(i)} \in V_{\tilde{\alpha}} \setminus \{\mathsf{src}, \mathsf{snk}\}$ corresponds to the same $\chi_{(i)}$ in $\tilde{\alpha}$. Thus, if $G_{\tilde{\alpha}}$ satisfies a condition i for non-determinism, α satisfies the same condition i in Definition 14, and vice versa.

Complexity: Given a regex α, let n denote the number of occurrences of terminals and variable references in α. We examine two steps of the computation: Computing $G_{\tilde{\alpha}}$, and converting it to $\mathcal{M}(\alpha)$.

For the first step, observe that $G_{\tilde{\alpha}}$ has $n + 2$ nodes, and if α is deterministic, each node has at most $\min(n, |\Sigma|)$ outgoing edges, which means that we can bound this number with $|\Sigma|$. Hence, if α is deterministic, $G_{\tilde{\alpha}}$ can be computed in time $O(|\Sigma||\alpha|n)$ by directly following the recursive definition of $G_{\tilde{\alpha}}$: If α is represented as a tree, it has at most $|\alpha|$ nodes, which means that the recursive rules have to be applied $O(|\alpha|)$ times. Each rule application requires the creation of at most $O(|\Sigma|n)$ edges, each of which uses a concatenation.

For the conversion, we need to process each edge $(u, \nu, v) \in E_{\tilde{\alpha}}$, and compute its function $\mathsf{net}_{\mathsf{unmark}(\nu)}$. From the recursive definition, we can immediately conclude that $|\nu| \in O(|\alpha|)$ (as α is deterministic, we do not even need to take into account that the definition for Kleene plus uses min). Hence, each edge can be turned into a transition in time $O(|\alpha)$. As α is deterministic, there are $O(|\Sigma|n)$ edges, which gives us a total time of $O(|\Sigma||\alpha|n)$ for this step.

As we have the same approximation for both steps, we conclude that the total running time is $O(|\Sigma||\alpha|n)$.

If α is not deterministic, this can be discovered during the construction, as soon as the recursive definition computes a non-deterministic memory occurrence graph $G_{\tilde{\beta}}$ for a non-deterministic subexpression β of α, or if hidden non-determinism is detected.

Unmarked edge labels: As mentioned above, if the goal is not to construct an $\mathcal{M}(\alpha)$ that is deterministic if and only if α is deterministic, but to turn every deterministic α in a deterministic $\mathcal{M}(\alpha)$ and to reject non-deterministic α, we can construct $G_{\tilde{\alpha}}$ by using unmarked ref-words on the labels. The only cases where using unmarked ref-words can hide non-determinism (in the sense that $G_{\tilde{\alpha}}$ is deterministic, although α is not) is in the rule for union. For example, consider $\alpha := \langle x \colon \varepsilon \rangle \vee \langle x \colon \varepsilon \rangle$, which satisfies condition 4 of Definition 14, as $\mathcal{L}(\tilde{\alpha})$ contains $[_{x(1)}]_{x(2)}$ and $[_{x(3)}]_{x(4)}$, due to $\tilde{\alpha} = ([_{x(1)}]_{x(2)}) \vee ([_{x(3)}]_{x(4)})$. If we use unmarked ref-words, $G_{\tilde{\alpha}}$ consists only of a single edge from src to snk with label $[_x]_x$, which is clearly deterministic. Nonetheless, we can detect this hidden non-determinism when recursively constructing $G_{\tilde{\alpha}}$, by checking whether there exist edges $(\mathsf{src}, \nu_1, \mathsf{snk}) \in E_{\tilde{\beta}}$ and $(\mathsf{src}, \nu_2, \mathsf{snk}) \in E_{\tilde{\gamma}}$ with $\nu_1 \neq \varepsilon$ or $\nu_2 \neq \varepsilon$. Hence, if the conversion algorithm encounters this case, it can reject the regex as non-deterministic.

Note that concatenation cannot hide non-determinism: For $u \in V_{\tilde{\beta}}$ and $v \in V_{\tilde{\gamma}}$, define $N_u := \{\nu \mid (u, \nu, \mathsf{snk}) \in E_{\tilde{\beta}}\}$ and $N_v := \{\nu \mid (\mathsf{src}, \nu, v) \in E_{\tilde{\gamma}}\}$. Assume that at least one of the two sets N_u and N_v contains more than one element. Then $E_{\tilde{\alpha}}$ contains at least two edges from u to v, which means that $G_{\tilde{\alpha}}$ is not deterministic. Finally, Kleene star is also unaffected by this change, as the presence of any edge $(\mathsf{src}, \nu, \mathsf{snk})$ with $\nu \neq \varepsilon$ causes non-determinism regardless of whether ν is marked or not.

Furthermore, note that an implementation of this construction can also represent each label ν in reduced form as $\mathsf{net}_{\mathsf{min}(\nu)}$, if it ensures that no hidden non-determinism is present. ◀

A.10 Lemma 40 (for Example 20)

▶ **Lemma 40.** *Let $L := \{F_{4i+3} \mid i \geq 0\}$. Then $L = \mathcal{L}(\beta)$ holds for the deterministic regex*

$$\beta := \mathtt{a}\langle x_0 \colon \mathtt{b}\rangle\langle x_1 \colon \mathtt{a}\rangle\big(\langle x_2 \colon \&x_1\&x_0\rangle\langle x_3 \colon \&x_1\&x_0\&x_1\rangle\langle x_0 \colon \&x_3\&x_2\rangle\langle x_1 \colon \&x_3\&x_2\&x_3\rangle\big)^*.$$

Proof. First, we observe that $\mathcal{R}(\beta) = \{r_i \mid i \geq 0\}$, where the ref-words r_i are defined by

$$r_0 := \mathtt{a}[_{x_0}\mathtt{b}]_{x_0}[_{x_1}\mathtt{a}]_{x_1},$$
$$\hat{r} := [_{x_2}x_1x_0]_{x_2}[_{x_3}x_1x_0x_1]_{x_3}[_{x_0}x_3x_2]_{x_0}[_{x_1}x_3x_2x_3]_{x_1},$$

and $r_{i+1} := r_i \cdot \hat{r}$ for all $i \geq 0$. We now proof by induction that, for each $i \geq 0$, $\mathcal{D}(r_i) = F_{4i+3}$, and the rightmost values that are assigned to x_0 and x_1 are F_{4i} and F_{4i+1}, respectively. For $i = 0$, this is obviously true: $\mathcal{D}(r_0) = \mathtt{aba} = F_3$, x_0 is assigned $\mathtt{b} = F_0$, and x_1 is assigned $\mathtt{a} = F_1$.

Now assume that the claim holds for some $i \geq 0$, and consider r_i. Then we can observe that $\mathcal{D}(r_{i+1}) = \mathcal{D}(r_i \cdot \hat{r}) = \mathcal{D}(r_i) \cdot \mathcal{D}(s)$, where the ref-word s is obtained from $\hat{r}$ by replacing x_0 and x_1 with their respective values F_{4i} and F_{4i+1}. Hence,

$$\begin{aligned} s &= [_{x_2}F_{4i+1} \cdot F_{4i}]_{x_2}[_{x_3}F_{4i+1} \cdot F_{4i} \cdot F_{4i+1}]_{x_3}[_{x_0}x_3x_2]_{x_0}[_{x_1}x_3x_2x_3]_{x_1} \\ &= [_{x_2}F_{4i+2}]_{x_2}[_{x_3}F_{4i+3}]_{x_3}[_{x_0}x_3x_2]_{x_0}[_{x_1}x_3x_2x_3]_{x_1}. \end{aligned}$$

The second part of this equation uses that $F_{n+3} = F_{n+2} \cdot F_{n+1} = F_{n+1} \cdot F_n \cdot F_{n+1}$ holds for all $n \geq 0$. We now construct a ref-word t by replacing the variables x_2 and x_3 in s with their respective values. Hence,

$$t = [_{x_2} F_{4i+2}]_{x_2} [_{x_3} F_{4i+3}]_{x_3} [_{x_0} F_{4i+3} \cdot F_{4i+2}]_{x_0} [_{x_1} F_{4i+3} \cdot F_{4i+2} \cdot F_{4i+3}]_{x_1}$$
$$= [_{x_2} F_{4i+2}]_{x_2} [_{x_3} F_{4i+3}]_{x_3} [_{x_0} F_{4i+4}]_{x_0} [_{x_1} F_{4i+5}]_{x_1}.$$

Then $\mathcal{D}(r_{i+1}) = \mathcal{D}(r_i \cdot t)$ holds, and r_{i+1} assigns x_0 and x_1 as t does. Hence, x_0 is assigned $F_{4(i+1)}$, and x_1 is assigned $F_{4(i+1)+1}$, as required by the claim. To see that $\mathcal{D}(r_{i+1}) = F_{4(i+1)+3}$, we observe that

$$\begin{aligned}\mathcal{D}(r_{i+1}) &= \mathcal{D}(r_i \cdot t)\\ &= F_{4i+3} \cdot \mathcal{D}([_{x_2} F_{4i+2}]_{x_2} [_{x_3} F_{4i+3}]_{x_3} [_{x_0} F_{4i+4}]_{x_0} [_{x_1} F_{4i+5}]_{x_1})\\ &= F_{4i+3} \cdot F_{4i+2} \cdot F_{4i+3} \cdot F_{4i+4} \cdot F_{4i+5}\\ &= F_{4i+5} \cdot F_{4i+4} \cdot F_{4i+5}\\ &= F_{4i+7} = F_{4(i+1)+3}.\end{aligned}$$

This concludes the proof. ◀

Also note that we can construct an $M \in \mathsf{DTMFA}^{\mathsf{rej}}$ with $\mathcal{L}(M) = \{F_n \mid n \geq 0\}$ by making the right states of $\mathcal{M}(\beta)$ accepting. In particular, this causes the cycle in M to have multiple accepting states. The authors conjecture that this is unavoidable, and that $\mathcal{L}(\beta)$ is not a DRX-language.

A.11 Proof of Lemma 22

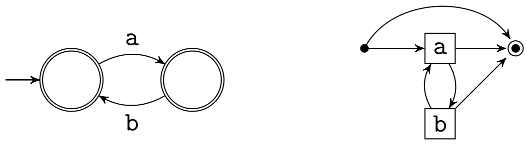
Proof. To show that $L \in \mathcal{L}(\mathsf{TMFA})$, we construct a regex α for L, by $\alpha := \alpha_1 \vee \alpha_2$, where $\alpha_1 := \mathtt{a}(\mathtt{a}^4)^*$, and α_2 is the deterministic regex with $\mathcal{L}(\alpha_2) = \{\mathtt{a}^{4^i} \mid i \geq 1\}$ from Example 21.

Next, observe that L contains the arithmetic progression $\{\mathtt{a}^{4i+1} \mid i \geq 0\}$, and $\overline{L} := \{\mathtt{a}\}^* \setminus L$ contains the arithmetic progression $\{\mathtt{a}^{4i+2} \mid i \geq 0\}$. Assume that L is a DTMFA-language. Then there is an $A \in \mathsf{DTMFA}^{\mathsf{rej}}$ that accepts L or $\overline{L}$. Then, by Lemma 10, L is regular (note that in case $\mathcal{L}(A) = \overline{L}$, we also use that the class of regular languages is closed under complementation). But as L is not regular, this is a contradiction. (To show that L is not regular, first assume the contrary. Then $\mathcal{L}(\alpha_2) = L \cap \{\mathtt{a}^4\}^*$ would be regular, as the class of regular languages is closed under intersection. But $\mathcal{L}(\alpha_2)$ is not regular, as for every pair $i \neq j$, $\mathtt{a}^{4^i}$ and $\mathtt{a}^{4^j}$ are not Nerode-equivalent. ◀

A.12 Proof of Lemma 23

Proof. Before we assume the existence of an $\alpha \in \mathsf{DRX}$ with $\mathcal{L}(\alpha) = L$ (and use this to obtain a contradiction), we first examine the structure of any $M \in \mathsf{DTMFA}^{\mathsf{rej}}$ with $\mathcal{L}(M) = L$. We observe that, with the exemption of states that are unreachable or cannot reach an accepting state, M must consist of a chain (which might be empty) that is followed by a cycle that contains at least one final state (like a DFA for a unary language, see the proof of Theorem 25, in particular the picture).

This is for the following reason: First, like for every DTMFA, each state of M that has an outgoing memory recall transition cannot have any other outgoing transitions. The same holds for ε-transitions. Furthermore, due to the structure of L, in M no state can have an outgoing transition that consumes $\mathtt{a}$ and an outgoing transition that consumes $\mathtt{b}$ at the same time. For DTMFA, this is not problematic. In fact, as L is regular, we can interpret any DFA for L as a DTMFA for L. For example, consider the following minimal incomplete DFA for L, and its corresponding notation as an occurrence graph (without markings):



If we consider the occurrence graph notation, we see that this DFA cannot be obtained from a deterministic regular expression (at least not using the Glushkov construction, which – in the absence of variables – is identical to the construction from the proof of Theorem 17). Note that the states for a and b belong to the same strongly connected component. Hence, if there is an $\alpha \in$ DREG such that this automaton is $\mathcal{M}(\alpha)$, then α must contain a subexpression β^+ with the occurrences $\mathtt{a}_{(i)}$ and $\mathtt{b}_{(j)}$ that correspond to these states. Then $G_{\tilde{\beta}}$ must contain edges from $\mathtt{a}_{(i)}$ and $\mathtt{b}_{(j)}$ to snk, and from src to $\mathtt{a}_{(i)}$. Using the rule for Kleene plus from the proof of Theorem 17, we see that $G_{\tilde{\alpha}}$ must contain an edge from $\mathtt{a}_{(i)}$ to itself. This is a contradiction. Of course, this argument only shows that this DFA cannot be obtained from a deterministic regular expression; but it can be generalized to show that there is no $\alpha \in$ DREG with $\mathcal{L}(\alpha) = L$ (see e. g. [9], and in particular [12], which explains how to apply the technique from [9] on this language).

We shall now use a similar line of reasoning to obtain a contradiction from the assumption that there is an $\alpha \in$ DRX with $\mathcal{L}(\alpha) = L$. As explained above, the DTMFA $\mathcal{M}(\alpha)$ must consist of a chain and a cycle (by definition, each state of $\mathcal{M}(\alpha)$ is reachable; and from each state, we can reach an accepting state). This means that $G_{\tilde{\alpha}}$ contains a cycle $v_1, \ldots, v_n$ for some $n \geq 1$ and $v_i \in (\tilde{\Sigma} \cup \tilde{\Xi})$ such that there is an edge from v_i to v_{i+1} for $1 \leq i < n$ and from v_n to v_1. Hence, each v_i has at most two outgoing edges: One to the next node in the cycle, and (if it is an accepting state) one to the sink node snk. Furthermore, exactly one v_i has an incoming edge from outside the cycle, let this be v_1.

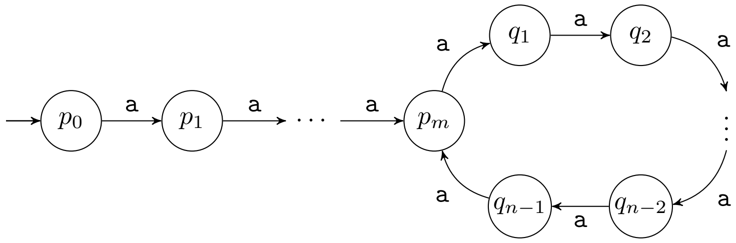
From the construction of $G_{\tilde{\alpha}}$, this cycle must have been generated from a Kleene plus in α. But this allows us to conclude that only exactly one v_i can have an edge to snk; and furthermore, that this must be v_n. This can be concluded from the following reasoning: If there existed nodes v_i, v_j with $i \neq j$, and both have an edge to snk, then the construction for Kleene plus would require edges from both v_i and v_j to v_1, which would break the cycle structure. Likewise, if $i \neq n$, then there must be an edge from v_i to v_1, and from v_i to v_{i+1}, which is a contradiction to our previous observations.

Hence, each iteration of the cycle must consume exactly one terminal letter (otherwise, we would skip over words of L), alternating between a and b. Thus, $v_i \in \tilde{\Xi}$ must hold for all $1 \leq i \leq n$ (as $v_i = a_{(j)}$ with $a_{(j)} \in \tilde{\Sigma}$ would consume a in every iteration). Assume that we enter an iteration that consumes a (the same reasoning shall hold for b). Then no variable that is recalled can contain b, and no variable can be bound to b, as otherwise, the iteration would consume more than a. But in the next iteration, the same variables are recalled, and as neither of them contains b, the iteration cannot consume b. Therefore, we arrive at a contradiction, and conclude that there is no $\alpha \in$ DRX with $\mathcal{L}(\alpha) = L$. ◀

A.13 Proof of Theorem 25

Proof. Assume that $L \subseteq \{\mathtt{a}\}^*$. Our goal is to construct an $\alpha \in$ DRX with $\mathcal{L}(\alpha) = L$. For technical reasons, we assume that $\varepsilon \notin L$ (this is no problem, as for any $\alpha \in$ DRX with $\varepsilon \notin \mathcal{L}(\alpha)$, $(\alpha \vee \varepsilon) \in$ DRX). If L is finite, $L \in \mathcal{L}$(DREG), and hence $L \in \mathcal{L}$(DRX). (As we shall see, our construction can also be used for finite languages, by replacing α_{cycle} in $\alpha_k^{\mathsf{chain}}$ below with ε. But to streamline the argument, we only consider infinite L.)

Let M be a DFA with $\mathcal{L}(M) = L$. Assume that all states of M are reachable, and that from each state, an accepting state can be reached. Then M has the following shape, (in this picture, we do not distinguish between accepting and non-accepting states):



We refer to the states p_0 to p_m as the *chain*, and to the states $q_1, \ldots, q_{n-1}$ and p_m as the *cycle*. Without loss of generality, we can assume that p_m is accepting (the cycle contains at least an accepting state; and as the automaton does not have to be minimal, we can extend the chain of p_i by unrolling the cycle until it starts with an accepting state).

Now there exists a $k \geq 1$ and $c_1, \ldots, c_k \geq 0$ such that the words $\mathtt{a}^{c_1}, \mathtt{a}^{c_1+c_2}, \ldots, \mathtt{a}^{c_1+\cdots+c_k}$ are exactly the words that are accepted in the chain (recall that we assume that p_m is accepting, and that p_0 is not accepting, as $\varepsilon \notin L$). Furthermore, there exists an $\ell \geq 1$ and $b_1, \ldots, b_\ell \geq 1$ such that the words $\mathtt{a}^{b_1}, \mathtt{a}^{b_1+b_2}, \ldots, \mathtt{a}^{b_1+\cdots+b_\ell}$ are exactly the words that advance the cycle from p_m to each of the accepting states. These conditions also imply $m = \sum_{i=1}^{k} c_i$ and $n = \sum_{i=1}^{\ell} b_i$.

As an additional restriction, we assume that $b_1 \geq 2$. This is possible for the following reasons: If $b_i = 1$ for all i, we can replace the cycle with the deterministic regular expression $\mathtt{a}^*$ and are done. Furthermore, if $b_1 = 1$, but there is a $b_i \geq 2$, we can unroll the cycle into the chain until $b_1 \geq 2$ (for the "new" b_1).

We define $\alpha := \alpha_1^{\mathsf{chain}}$, where, for $1 \leq i < k$,

$$\alpha_i^{\mathsf{chain}} := \mathtt{a}^{c_i}(\varepsilon \vee \alpha_{i+1}^{\mathsf{chain}}),$$
$$\alpha_k^{\mathsf{chain}} := \langle x_\ell \colon \mathtt{a} \rangle \mathtt{a}^{c_k - 1}(\varepsilon \vee \alpha_{\mathsf{cycle}}).$$

Before we define α_{cycle}, note that if we disregard the words that can be generated by the subexpression α_{cycle}, $\mathcal{L}(\alpha)$ contains exactly the words that are accepted by the chain. Furthermore, if we assume that α_{cycle} is a deterministic regex and that its language does not contain ε, we can conclude $\alpha \in \mathsf{DRX}$. Finally, note that if we first enter α_{cycle}, the variable x_ℓ contains $\mathtt{a}$.

The central part of the construction is defining α_{cycle} in such a way that it simulates the cycle by generating exactly the words $\mathtt{a}^{b_1}, \mathtt{a}^{b_1+b_2}, \ldots, \mathtt{a}^{b_1+\cdots+b_\ell}$. We define

$$\alpha_{\mathsf{cycle}} := \big(\alpha_{\mathsf{shift}} \cdot \alpha_{\mathsf{cont}}\big)^+,$$
$$\alpha_{\mathsf{shift}} := \langle x_0 \colon \& x_\ell \rangle \langle x_\ell \colon \& x_{\ell-1} \rangle \cdots \langle x_2 \colon \& x_1 \rangle \langle x_1 \colon \& x_0 \rangle,$$
$$\alpha_{\mathsf{cont}} := \& x_1^{b_1 - 2} \cdot \& x_2^{b_2 - 1} \cdot \cdots \cdot \& x_\ell^{b_\ell - 1}.$$

The idea behind this definition is as follows: Before the first iteration of the Kleene plus, x_ℓ contains $\mathtt{a}$, and all other variables default to ε. Now, note that passing through α_{shift} shifts the $\mathtt{a}$ from x_ℓ to x_1, or from x_i to x_{i+1} for $1 \leq i < \ell$. As this cyclic shift needs an extra variable, if x_1 is set to $\mathtt{a}$, x_0 is also set to $\mathtt{a}$ (which is overwritten in the next iteration of the plus).

Hence, in the i-th iteration of the plus, the variable x_j with $j := ((i-1) \bmod \ell) + 1$ is set to $\mathtt{a}$, and if $j = 1$, then x_0 is also set to $\mathtt{a}$. All other variables are set to ε. This means that

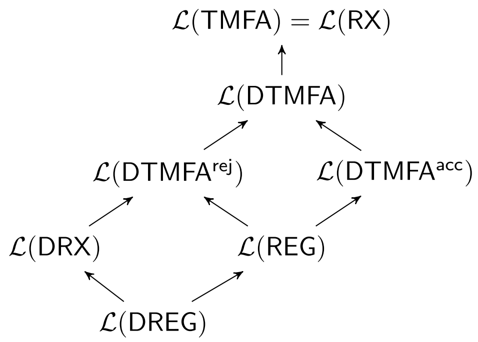


Figure 3 An unflattened version of Figure 1.

α_{shift} produces $\mathtt{a}^2$ in the i-th iteration if $(i \bmod \ell) = 1$, and $\mathtt{a}$ in all other iterations. This is the reason we ensured that $b_1 \geq 2$ above, as we now use α_{cont} to produce the remaining letters. If $j = 1$, then the $\& x_1^{b_1 - 2}$ in α_{cont} produces $\mathtt{a}^{b_1-2}$, which means that in this iteration, $\alpha_{\mathsf{shift}} \cdot \alpha_{\mathsf{cont}}$ produces $\mathtt{aa} \cdot \mathtt{a}^{b_1-2} = \mathtt{a}^{b_1}$ (recall that all other variables are set to ε). If $j > 1$, then α_{cont} used $\& x_j^{b_j - 1}$ to produce $\mathtt{a}^{b_j-1}$, which means that $\alpha_{\mathsf{shift}} \cdot \alpha_{\mathsf{cont}}$ produces $\mathtt{a} \cdot \mathtt{a}^{b_j-1} = \mathtt{a}^{b_j}$.

In conclusion, the i-th iteration of the Kleene plus in α_{cycle} adds the word $\mathtt{a}^{b_j}$ for $j := ((i-1) \bmod \ell) + 1$; which means that α_{cycle} simulates the cycle. Hence, $\mathcal{L}(\alpha) = L$. As α_{cycle} contains no disjunctions or Kleene plus (except for the surrounding plus), it is deterministic. As remarked above, this allows us to conclude $\alpha \in \mathsf{DRX}$. ◀

A.14 Figure 3 and Proof of Theorem 26

Proof. This follows from our previous observations as follows:

1. $\mathcal{L}(\mathsf{DREG}) \subset \mathcal{L}(\mathsf{DRX})$: The inclusion follows from the fact that our definition of determinism for regex is an extension of the notion of determinism for proper regular expressions. To see that the inclusion is proper, we recall any of the non-regular DRX-languages that we have seen, for example $\{w\mathtt{c}w \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$ and $\{\mathtt{a}^{n^2} \mid n \geq 0\}$ from Example 15.
2. $\mathcal{L}(\mathsf{DRX}) \subset \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$: The inclusion follows from Theorem 17, it is proper due to the language $\{(\mathtt{ab})^{\frac{1}{2}i} \mid i \geq 0\}$, see Lemma 23.
3. $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \subset \mathcal{L}(\mathsf{DTMFA})$: The inclusion holds by definition. It is proper as the inclusion $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}}) \subset \mathcal{L}(\mathsf{DTMFA})$ also holds by definition, and as $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ and $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$ are incomparable (see below).
4. $\mathcal{L}(\mathsf{DTMFA}) \subset \mathcal{L}(\mathsf{TMFA})$: Again, the inclusion holds by definition. Languages that separate the two classes are for example the language of all ww (where w is from a non-unary alphabet, see Example 8), and the language $\{\mathtt{a}^{4i+1} \mid i \geq 0\} \cup \{\mathtt{a}^{4^i} \mid i \geq 1\}$ from Lemma 22.
5. $\mathcal{L}(\mathsf{TMFA}) = \mathcal{L}(\mathsf{RX})$ is the statement of Theorem 4.
6. $\mathcal{L}(\mathsf{DRX})$ and $\mathcal{L}(\mathsf{REG})$ are incomparable: Again, we can use $\{(\mathtt{ab})^{\frac{1}{2}i} \mid i \geq 0\}$ from Lemma 23, and a non-regular DRX-language, like $\{w\mathtt{c}w \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$.
7. $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$ and $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$ are incomparable: Due to Proposition 12, over a unary alphabet, for every non-regular language $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$, we have $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \setminus \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$, and $\overline{L} \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}}) \setminus \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$. Hence, we can choose e.g. $L = \{\mathtt{a}^{n^2} \mid n \geq 0\}$ and its complement to show the two classes to be incomparable.

8. $\mathcal{L}(\mathsf{DRX})$ and $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$ are incomparable: We first note that, since $\mathcal{L}(\mathsf{REG}) \subseteq \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$, the language $\{(\mathtt{ab})^{\frac{1}{2}i} \mid i \geq 0\}$ is in $\mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$, but, due to Lemma 23, not in $\mathcal{L}(\mathsf{DRX})$. Moreover, $L = \{\mathtt{a}^{n^2} \mid n \geq 0\} \in \mathcal{L}(\mathsf{DRX})$ (see Example 15), but if $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$, then, due to Theorem 17, also $L \in \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}}) \cap \mathcal{L}(\mathsf{DTMFA}^{\mathsf{acc}})$, which, by Proposition 12, leads to the contradiction $L \in \mathcal{L}(\mathsf{REG})$.

This concludes the proof. ◀

A.15 Proof of Theorem 27

Proof. The proofs for the operations that apply to both classes follow the same basic scheme: We start with one (or more) DRX-language(s), and show that applying the operation yields a language that is not a $\mathsf{DTMFA}^{\mathsf{rej}}$-language:

Union: We use $L_1 := \{\mathtt{a}^{4i+1} \mid i \geq 0\}$ and $L_2 := \{\mathtt{a}^{4^i} \mid i \geq 1\}$, which are defined by the deterministic regex $\alpha_1 := \mathtt{a}(\mathtt{a}^4)^*$, and $\alpha_2 := \mathtt{a}^2 \cdot \langle x\colon \mathtt{a}^2 \rangle \cdot \big(\langle y\colon \&x \cdot \&x \rangle \cdot \langle x\colon \&y \cdot \&y \rangle\big)^*$, see Example 21. Then $L_1 \cup L_2 \notin \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$, as shown in Lemma 22.

Concatenation: Define the deterministic regexes $\alpha_3 := \mathtt{a}^+$ and $\alpha_4 := \langle x\colon \mathtt{a}^* \rangle \cdot \mathtt{b} \cdot \&x$. Then $\mathcal{L}(\alpha_4) = \{\mathtt{a}^i\mathtt{b}\mathtt{a}^i \mid i \geq 0\}$, and $\mathcal{L}(\alpha_3) \cdot \mathcal{L}(\alpha_4) = \{\mathtt{a}^i\mathtt{b}\mathtt{a}^j \mid i > j \geq 0\}$. As shown in Example 9, $\mathcal{L}(\alpha_3) \cdot \mathcal{L}(\alpha_4)$ is not a $\mathsf{DTMFA}^{\mathsf{rej}}$-language.

Reversal: Let $\alpha_5 := \langle x\colon \mathtt{a}^* \rangle \cdot \mathtt{b} \cdot \&x \cdot \mathtt{a}^+$. Then $\alpha_5 \in \mathsf{DRX}$, and $\mathcal{L}(\alpha_5) = \{\mathtt{a}^j\mathtt{b}\mathtt{a}^i \mid i > j \geq 0\}$. Reversing $\mathcal{L}(\alpha_5)$ again gives us the language from Example 9.

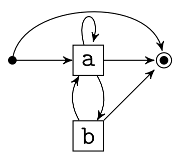
Complement: This follows directly from Proposition 12. Consider e. g. $\{\mathtt{a}^{n^2} \mid n \geq 0\}$.

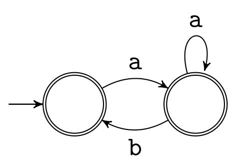
Homomorphism: Let $\alpha_6 := ((\mathtt{c} \cdot \alpha_1) \vee (\mathtt{d} \cdot \alpha_2))$. Then $\alpha_6 \in \mathsf{DRX}$, but $h(\mathcal{L}(\alpha_6)) = L_1 \cup L_2$ for the morphism h that is defined by $h(x) := x$ if $x \in \{\mathtt{a}, \mathtt{b}\}$ and $h(x) := \varepsilon$ if $x \in \{\mathtt{c}, \mathtt{d}\}$.

Inverse homorphism: Define a morphism g by $g(\mathtt{a}) := g(\mathtt{b}) := \mathtt{a}$, and $g(\mathtt{c}) := \mathtt{b}$. Then let $L_7 := g^{-1}(\mathcal{L}(\alpha_4)) = \{u \cdot \mathtt{c} \cdot v \mid u, v \in \{\mathtt{a}, \mathtt{b}\}^*, |u| = |v|\}$. We use Lemma 7 to show that $L_7 \notin \mathcal{L}(\mathsf{DTMFA}^{\mathsf{rej}})$. Assume to the contrary that it is, and choose $m \geq 1$. Then there exist $n \geq m$ and words p_n, v_n that satisfy the conditions of Lemma 7. We now distinguish the following cases: First, assume that p_n does not contain the letter $\mathtt{c}$. Then choose a $d \in \{\mathtt{a}, \mathtt{b}\}$ that is not the first letter of v_n, and define $u := d \cdot \mathtt{c} \cdot \mathtt{a}^{|p_n|+1}$. Then $p_n u \in L_7$; but as v_n is not a prefix of u, this contradicts Lemma 7. Now assume that p_n contains $\mathtt{c}$. Then $p_n = w_1 \mathtt{c} w_2$ with $w_1, w_2 \in \{\mathtt{a}, \mathtt{b}\}^*$, and $|w_1| \geq |w_2| + n \geq |w_2| + m$. Again, choose $d \in \{\mathtt{a}, \mathtt{b}\}$ such that d is not the first letter of v_n, and define $u := d \cdot \mathtt{a}^{|w_1|-|w_2|-1}$. Then $p_n u = w_1 \mathtt{c} w_3$ for $w_3 = w_2 d \cdot \mathtt{a}^{|w_1|-|w_2|-1}$, and $|w_3| = |w_1|$. Hence, $p_n u \in L_7$, but as v_n is not a prefix of u, this contradicts Lemma 7.

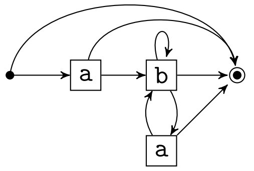
Intersection: In order to show both claims on the intersection of $\mathcal{L}(\mathsf{DRX})$, it suffices to show that we can obtain a language that is not a DRX-language by intersecting two deterministic regular languages. Accordingly, we define deterministic regular expressions $\beta_1 := (\mathtt{a}(\mathtt{b} \vee \varepsilon))^*$ and $\beta_2 := \varepsilon \vee \big(\mathtt{a}(\mathtt{b}(\mathtt{a} \vee \varepsilon))^*\big)$ (these expressions have been obtained by very minor modifications to the expressions that Caron, Han, Mignot [12] use to show that $\mathcal{L}(\mathsf{DREG})$ is not closed under intersection).

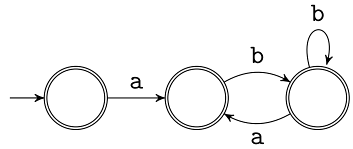
Let $L_8 = \{(\mathtt{ab})^{\frac{1}{2}i} \mid i \geq 0\}$. To show that $L_8 = \mathcal{L}(\beta_1) \cap \mathcal{L}(\beta_2)$, we follow the approach from [12], and first consider $\mathcal{M}(\beta_1)$ and the corresponding minimal incomplete DFA:





Likewise, we consider $\mathcal{M}(\beta_2)$ and the corresponding minimal incomplete DFA (which merges the two states for a):





Now it is easily seen that $L_8 = \mathcal{L}(\beta_1) \cap \mathcal{L}(\beta_2)$. From Lemma 23, we know that $L_8 \notin \mathcal{L}(\mathsf{DRX})$. Hence, the class of deterministic regex languages is not closed under intersection with deterministic regular languages, which also implies that it is not closed under intersection. ◀

A.16 Proof of Theorem 28

Proof. We show this with a reduction from Post's Correspondence Problem (PCP). Let $(u_1, v_1), \ldots, (u_k, v_k) \in \Sigma^* \times \Sigma^*$, $k \geq 1$, be a PCP instance. Our goal is to construct $\alpha, \beta \in \mathsf{DRX}$ such that $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta) \neq \emptyset$ if and only if there exists a sequence $i_1, \ldots, i_n$, $n \geq 1$ and $1 \leq i_j \leq k$, such that $u_{i_1} \cdots u_{i_n} = v_{i_1} \cdots v_{i_n}$. To do so, we first introduce an alphabet $A := \{a_1 \ldots, a_k\}$ such that A, Σ, and $\{\#, \$, ¢\}$ are pairwise disjoint (at the end of the proof, we discuss how this construction can be adapted to binary terminal alphabets). We then define

$$\alpha := \Big(\bigvee_{i=1}^{k} a_i \,\#\, u_i \langle x \colon \Sigma^* \rangle \,\#\, v_i \langle y \colon \Sigma^* \rangle \,\$\, \& x \,\#\, \& y \,¢\Big)^*,$$

$$\beta := A \,\#\langle z \colon \Sigma^+ \rangle \,\#\, \& z \,\$\Big(\langle x \colon \Sigma^+ \rangle \,\#\langle y \colon \Sigma^+ \rangle \,¢\, A \,\#\, \& x \,\#\, \& y \,\$\Big)^* \,\#\, ¢\,.$$

To see that α is deterministic, note that the disjunction ranges over the letters from A. For β, we observe that after each iteration of the starred subexpression, we read either a letter from Σ, and start a new iteration, or $\#$, which means that this was the last iteration.

We now claim that $w \in \mathcal{L}(\alpha) \cap \mathcal{L}(\beta)$ if and only if there exist an $n \geq 1$ and $i_1, \ldots, i_n$ with $1 \leq i_j \leq k$ such that $u_{i_1} \cdots u_{i_n} = v_{i_1} \cdots v_{i_n}$ and $w = w_1 \,¢\, w_2 \,¢ \cdots w_n \,¢$, where

$$w_j = a_{i_j} \,\#\, u_{i_j} \cdots u_{i_n} \,\#\, v_{i_j} \cdots v_{i_n} \,\$\, u_{i_{j+1}} \cdots u_{i_n} \,\#\, v_{i_{j+1}} \cdots v_{i_n}.$$

Take note that w_n always ends on $\$\,\#$. Informally explained, w encodes how a solution of the PCP instance is constructed, where the finished solution is in w_1, and the start of the construction is at w_n. Starting at w_1, the sequence of w_j can be understood as splitting off pairs of prefixes (u_j, v_j) from the solution, where each word w_j also encodes which tuple (u_j, v_j) is processed (by using the preceding symbol a_j as a marker), and the words before and after the pair is split off (to the left and right of \$, respectively).

Here, α ensures that in each w_j, u_j and v_j are split off correctly, while β ensures that the "after" words of w_j are the "before" words of w_{j+1}. Hence, such a w exists if and only if the instance of the PCP has a solution. As the existence of the latter is undecidable (see e. g. Hopcroft and Ullman [32]), deciding $\mathcal{L}(\alpha) \cap \mathcal{L}(\beta) \neq \emptyset$ is also undecidable.

To adapt the construction to a binary alphabet (say, $\{\mathtt{a}, \mathtt{b}\}$), we use a morphism $h \colon (A \cup \Sigma \cup \{\#, \$, ¢\})^* \to \{\mathtt{a}, \mathtt{b}\}^*$ that is defined as follows:

- $h(a_i) := \mathtt{ab}^i\mathtt{a}$ for all $a_i \in A$,
- $h(b_i) := \mathtt{ab}^i\mathtt{a}$, where we assume an arbitrary ordering on Σ with $\Sigma = \{b_1, \ldots, b_{|\Sigma|}\}$,

- $h(\#) := \texttt{bab}$, $h(\$) := \texttt{ba}^2\texttt{b}$, and $h(¢) := \texttt{ba}^3\texttt{b}$.

If we apply h to α and β by applying h to each terminal, we obtain regex $h(\alpha)$ and $h(\beta)$ such that $\mathcal{L}(h(\alpha)) \cap \mathcal{L}(h(\beta)) \neq \emptyset$ if and only if the instance of the PCP has a solution. The only problem is that these regex are not deterministic, as there are disjunctions that start with the same terminal letter. But each of these disjunctions can be rewritten into a deterministic disjunction by nesting the branches. For example, consider the disjunction $(a_1 \vee a_2 \vee a_3)$. Using h, this becomes $(\texttt{aba} \vee \texttt{ab}^2\texttt{a} \vee \texttt{ab}^3\texttt{a})$, which is not deterministic, but can be rewritten to the equivalent $(\texttt{ab}(\texttt{a} \vee (\texttt{b}(\texttt{a} \vee \texttt{ba}))))$.

Now, note that if we apply this rewriting to the disjunctions to $h(\alpha)$ and $h(\beta)$ (including the disjunctions that are hidden in shorthand notations A and Σ), we obtain deterministic regex. In particular, note that Kleene plus and Kleene star are only used on elements of $A \cup \Sigma$, and are always followed by either $\#$ or ¢. As the encodings of the former start with a, while the encodings of the latter start with b, rewriting the disjunctions is enough to ensure determinism. ◀

A.17 Proof of Theorem 29

This proof combines ideas from two proofs from [24][3], the aforementioned intersection emptiness problem for vstar-free regex, and the conversion of so-called *vset automata* into formulas of SpLog, a fragment of $\mathsf{EC}^{\mathsf{reg}}$. We briefly discuss why these results cannot be used directly: First, note that we could use that the intersection emptiness problem for vstar-free regex is PSPACE-complete to conclude that the same problem for $\mathsf{TMFA}_{\mathsf{mcf}}$ is decidable, by converting each $M_i \in \mathsf{TMFA}$ into an equivalent vstar-free regex. But the best known trade-off in this direction is exponential[4], which make this approach unsuitable for a PSPACE upper bound. Second, we address the issue of vset-automata. In principle, a vset-automaton can be understood as a TMFA that has no memory recalls, but simulates these with added string equivalence predicates (basically, the relation of vset-automata to $\mathsf{TMFA}_{\mathsf{mcf}}$ is analogous to the relation of regex formulas to $\mathsf{RX}_{\mathsf{vsf}}$ that is described in [24]). Hence, the conversion of vset-automata and to $\mathsf{EC}^{\mathsf{reg}}$ from [24] could be used to convert $\mathsf{TMFA}_{\mathsf{mcf}}$ to $\mathsf{EC}^{\mathsf{reg}}$, but with one important caveat: In the runs of a vset-automaton, each variable can only be opened and closed exactly once. In contrast to this, a $\mathsf{TMFA}_{\mathsf{mcf}}$ may act multiple times on the same variable. In order to use the conversion of vset-automata to $\mathsf{EC}^{\mathsf{reg}}$ for $\mathsf{TMFA}_{\mathsf{mcf}}$, we would first need to convert the $\mathsf{TMFA}_{\mathsf{mcf}}$ to normal form (see Section A.3), which – due to the exponential blowup – is not possible in a PSPACE-algorithm. Dealing with this issue requires extra effort; but as we only want to decide intersection emptiness (instead of converting $\mathsf{TMFA}_{\mathsf{mcf}}$ to $\mathsf{EC}^{\mathsf{reg}}$-formulas), we can also simplify this constructions by just guessing a class of accepting runs, instead of building formulas for all accepting runs. Hence, the proof combines main idea of the proof of the conversion of $\mathsf{RX}_{\mathsf{vsf}}$ to $\mathsf{EC}^{\mathsf{reg}}$ (Theorem 28 in [24]) with the conversion of vset-automata (Theorem 21 in [24]), while including additional modifications for the more complicated memory actions of vset-automata, and at the same time avoiding the effort that is needed to convert an $M \in \mathsf{TMFA}_{\mathsf{mcf}}$ into an equivalent formula

[3] The full version of [24] with all proofs is available at `http://ddfy.de/sci/splog.pdf`

[4] Although the authors are not aware of a proof for the lower bound, the well-known exponential blowup from NFA to REG suggests that this applies. On the other hand, there is a non-recursive trade-off from $\mathsf{RX}_{\mathsf{vsf}}$ to REG (cf. [23]). Hence, in principle, it might be possible to counter-act the blowup from $\mathsf{TMFA}_{\mathsf{mcf}}$ to $\mathsf{RX}_{\mathsf{vsf}}$ by using variables, although this seems highly unlikely, and would probably not work in polynomial time (which we require to achieve a PSPACE upper bound). Either way, we conclude that based on current knowledge, we cannot directly use the algorithm for $\mathsf{RX}_{\mathsf{vsf}}$ intersection emptiness to decide intersection emptiness for $\mathsf{TMFA}_{\mathsf{mcf}}$ in PSPACE.

(by non-deterministically guessing one of finitely many sub-languages of $\mathcal{L}(M)$)[5].

Proof. We begin with the first lower bound: As shown by Martens, Neven, and Schwentick [40] (Theorem 3.10), the intersection emptiness problem for deterministic regular expressions is PSPACE-complete (if $|\Sigma|$ is not bounded; the paper does not discuss the unbounded case, and the proof cannot be adapted directly). This problem is defined as follows: Given $\beta_1, \dots, \beta_n \in \mathsf{DREG}$ for some $n \geq 2$, is $\bigcap_{i=1}^{n} \mathcal{L}(\beta_i) = \emptyset$? We use a new terminal letter $\# \notin \Sigma$, and define

$$\alpha := \langle x\colon \Sigma^* \rangle \, \# \big(\&x \, \#\big)^{n-1}$$
$$\beta := \beta_1 \, \# \, \beta_2 \, \# \cdots \beta_n \, \# \, .$$

First, observe that α and β are deterministic, as $\# \notin \Sigma$. Now $w \in \mathcal{L}(\alpha)$ holds if and only if $w = \big(\hat{w}\,\#\big)^n$ for some $\hat{w} \in \Sigma^*$, and $w \in \mathcal{L}(\beta)$ if and only there exist $w_1, \dots, w_n \in \Sigma^*$ with $w_i \in \mathcal{L}(\beta_i)$ and $w = w_1 \, \# \, w_2 \, \# \cdots w_n \, \#$. Hence, $(\mathcal{L}(\alpha) \cap \mathcal{L}(\beta)) \neq \emptyset$ if and only if $\bigcap_{i=1}^{n} \mathcal{L}(\beta_i) \neq \emptyset$. As this problem is PSPACE-complete, deciding $(\mathcal{L}(\alpha) \cap \mathcal{L}(\beta)) \stackrel{?}{=} \emptyset$ is PSPACE-hard.

The second lower bound is a reduction from the intersection emptiness problem for DFA, which is defined as follows: Given $M_1, \dots, M_n \in \mathsf{DFA}$ with $n \geq 2$, is there a $w \in \Sigma^*$ with $w \in \mathcal{L}(M_i)$ for all $1 \leq i \leq n$? This problem is PSPACE-complete (cf. Kozen [35]). We take a new terminal symbol $\# \notin \Sigma$, define α as above, and choose M to be the DFA for the language $\mathcal{L}(M_1) \, \# \, \mathcal{L}(M_2) \, \# \cdots \# \, \mathcal{L}(M_n)$ (as $\#$ does not occur in the languages of the DFA, this is trivially possible). The reasoning continues as above; but as the DFA can be defined on a binary alphabet, this proof does not require an unbounded alphabet.

The upper bound takes more work, including further definitions. Our goal is to encode the intersection emptiness problem for $\mathsf{TMFA_{mcf}}$ in $\mathsf{EC^{reg}}$, the existential theory of concatenation with regular constraints, which we now introduce (for a more detailed definition and examples on $\mathsf{EC^{reg}}$, see for example Freydenberger [24]).

One of the basic elements of $\mathsf{EC^{reg}}$-formulas are word equations: A *pattern* is a word $\alpha \in (\Sigma \cup \Xi)^*$, and a *word equation* is a pair of patterns (η_L, η_R), which can also be written as $\eta_L = \eta_R$ (hence the name equation). A *pattern substitution* is a homomorphism $\sigma\colon (\Xi \cup \Sigma)^* \to \Sigma^*$ with $\sigma(a) = a$ for all $a \in \Sigma$. It is a *solution* of a word equation (η_L, η_R) if $\sigma(\eta_L) = \sigma(\eta_R)$, and we write this as $\sigma \models (\eta_L, \eta_R)$. Less formally, a pattern substitution replaces all variables with terminal words (where multiple occurrences of the variable have to be substituted in the same way), and it is a solution of an equation if both sides have the same terminal word as a result.

The other basic building block are *constraint symbols*: For every ε-NFA A and every $x \in \Xi$, we can use a constraint symbol $\mathsf{C}_A(x)$. A pattern substitution σ satisfies $\mathsf{C}_A(x)$ if $\sigma(x) \in \mathcal{L}(A)$. We write this as $\sigma \models \mathsf{C}_A(x)$.

The *existential theory of concatenation with regular constraints* $\mathsf{EC^{reg}}$ is obtained by combining word equations and constraint symbols using $\wedge$, $\vee$ and existential quantification over variables. Semantics are defined canonically: We have $\sigma \models (\varphi_1 \wedge \varphi_2)$ if $\sigma \models (\varphi_1)$ and $\sigma \models \varphi_2$; and $\sigma \models (\varphi_1 \vee \varphi_2)$ if $\sigma \models (\varphi_1)$ or $\sigma \models \varphi_2$. Finally, $\sigma \models (\exists x\colon \varphi)$ if there

[5] We also avoid the extra effort that is needed to compute a SpLog-formula instead of an $\mathsf{EC^{reg}}$-formula; but compared to the aforementioned savings, this is negligible. Nonetheless, we do note that by combining all these proofs, it is possible to convert a $\mathsf{TMFA_{mcf}}$ into an equivalent SpLog-formula of polynomial size, which means that $\mathsf{TMFA_{mcf}}$ can be used for the core spanners of [20]. But this is outside the scope of the current paper.

exists a $w \in \Sigma^*$ such that $\sigma_{[x \to w]} \models \varphi$, where the pattern substitution $\sigma_{[x \to w]}$ is defined by $\sigma_{[x \to w]}(x) := w$, and $\sigma_{[x \to w]}(y) := \sigma(y)$ if $y \neq x$. In slight abuse of notation, we also write $w \models \varphi(x)$ if $\sigma \models \varphi(x)$ holds for the pattern substitution $\sigma(x) := w$.

For example, let $\varphi(x) := \exists y \colon \big((x = y\mathtt{b}y) \wedge \mathsf{C}_A(y)\big)$, where A is an NFA with $\mathcal{L}(A) = \{\mathtt{a}^*\}$. Then $w \models \varphi(x)$ if and only if $w = \mathtt{a}^n\mathtt{b}\mathtt{a}^n$ for some $n \geq 0$.

Given an $\mathsf{EC}^{\mathsf{reg}}$-formula φ, deciding the existence of a pattern substitution σ with $\sigma \models \varphi$ is PSPACE-complete, cf. Diekert[17].

We first prove the claim only for automata with rejecting trap states (as we shall see further down, the case for accepting memory failure states requires only a small modification). Before we proceed to the main idea of the construction, we first take a closer look at the accepting runs of memory-cycle-free TMFA.

Let $M \in \mathsf{TMFA}^{\mathsf{rej}}_{\mathsf{mcf}}$ with $M = (Q, \Sigma, \delta, q_0, F)$ and memories $\{1, \ldots, k\}$, and consider any accepting run of M. As M is memory cycle free, whenever it takes a memory transition from a state p to a state q, we know that p cannot occur anywhere else in the run. Otherwise, it would be possible to repeat the memory transition from p to q arbitrarily often, which would contradict the assumption that M is memory cycle free. Hence, we know that every accepting run of M can use at most $|Q| - 1$ memory transitions.

This allows us to condense any accepting run of M by considering only its memory transitions. Formally, for some $0 \leq \ell < |Q|$, we define a *condensed run* $\kappa = (\vec{q}, \vec{p}, \vec{\tau})$ of length ℓ as follows:

1. $\vec{q} = (q_0, \ldots, q_\ell)$ is a sequence of states, where q_0 is the starting state of M,
2. $\vec{p} = (p_0, \ldots, p_\ell)$ is a sequence of states, with $p_\ell \in F$,
3. $\vec{\tau} = (\tau_1, \ldots, \tau_\ell)$ is a sequence of memory transitions, where for each $1 \leq i \leq \ell$, either

   $$\tau_i = (p_i, x_i, q_{i+1}, s_{i,1}, \ldots, s_{i,k}) \text{ with } x_i \in \{1, \ldots, k\}, \text{ or}$$
   $$\tau_i = (p_i, b_i, q_{i+1}, s_{i,1}, \ldots, s_{i,k}) \text{ for some } b_i \in (\Sigma \cup \{\varepsilon\})$$

   and $s_{i,j} \in \{\mathtt{o}, \mathtt{c}, \mathtt{r}, \diamond\}$, $1 \leq j \leq k$. In the second case, we also require that there is at least one j with $s_{i,j} \neq \diamond$.
4. p_i is reachable from q_i without using memory transitions for all $0 \leq i \leq \ell$.

The intuition is that we condense the run to a sequence of states $(q_0, p_0, q_1, p_1, \ldots, q_\ell, p_\ell)$ that only contains the starting state q_0, a final state p_ℓ, and the states before and after each memory transition (as M runs from q_i to p_i only without memory transitions, and each memory transition τ_i takes the automaton from p_i to q_{i+1}).

As we shall see, each condensed run κ can be converted in polynomial time into an $\mathsf{EC}^{\mathsf{reg}}$ formula $\varphi_\kappa(w)$ that defines exactly the language of all $w \in \mathcal{L}(M)$ for which there is an accepting run of M that can be condensed to κ. In particular, the parts of the run between each pair of states q_i and p_i (which involve no memory transitions) shall be handled by appropriate regular constraints.

Now, given $M_1, \ldots, M_n \in \mathsf{TMFA}^{\mathsf{rej}}_{\mathsf{mcf}}$, we proceed as follows to decide whether $\bigcap_{i=1}^n \mathcal{L}(M_i) = \emptyset$. First, we guess a condensed run κ_i for each M_i (as the length of each sequence is bounded by the number of states of M_i and as PSPACE = NPSPACE, this is allowed). Next, we convert each κ_i into an $\mathsf{EC}^{\mathsf{reg}}$-formula $\varphi_i(w)$ (as we shall see, this is possible in polynomial time). Finally, we combine these into the formula $\varphi(w) := \bigwedge_{i=1}^n \varphi_i(w)$, and decide whether φ is satisfiable (as mentioned above, this is possible in PSPACE, see Diekert [17]). As φ is satisfiable if and only if there exists a $w \in \bigcap_{i=1}^n \mathcal{L}(M_i)$, this proves the claim for $\mathsf{TMFA}^{\mathsf{rej}}_{\mathsf{mcf}}$ (as mentioned above, we shall discuss the case of accepting failure states at the end of the proof).

We now discuss how to construct φ_κ from κ. Consider a word $w \in \mathcal{L}(M)$, and the condensed run κ of length ℓ for any accepting run of M on w. Then w can be decomposed into $w = u_0 v_1 u_1 \cdots v_\ell u_\ell$ with $u_i, v_i \in \Sigma^*$ such that u_i is the word that M consumes when processing from q_i to p_i (without using memory tranisitons), and v_i is the word that is consumed when processing from p_{i-1} to q_i (using the memory transition τ_i). This is illustrated by the following picture:

$$q_0 \xrightarrow{u_0} p_0 \xrightarrow[\tau_1]{v_1} q_1 \xrightarrow{u_1} p_1 \xrightarrow[\tau_2]{v_2} \cdots \xrightarrow{u_{\ell-1}} p_{\ell-1} \xrightarrow[\tau_\ell]{v_\ell} q_\ell \xrightarrow{u_\ell} p_\ell$$

Following this intuition, we define

$$\varphi_\kappa(w) := \exists u_0, \ldots, u_\ell, v_1, \ldots, v_\ell : \\ (w = u_0 v_1 u_1 \cdots v_\ell u_\ell) \wedge \bigwedge_{i=0}^{\ell} \mathsf{C}_{M_{q_i,p_i}}(u_i) \wedge \bigwedge_{i \in T} (v_i = b_i) \wedge \bigwedge_{i \in R} (v_i = \eta_i),$$

where the following holds:

1. for $p, q \in Q$, $M_{q,p}$ is the ε-NFA that is obtained from M by removing all memory transitions, using q as starting and p as only finite state,
2. $T \subseteq \{1, \ldots, \ell\}$ is the set of all i such that τ_i is not a memory recall (i. e., τ_i consumes b_i),
3. $R \subseteq \{1, \ldots, \ell\}$ is the set of all i such that τ_i is a memory recall (i. e., τ_i recalls x_i),
4. for each $i \in R$, we define η_i to describe the current content for x_i (as the memory actions are completely determined by the τ_j, this is directly possible by checking the τ_j with $j \leq i$). There are three possible cases:
   a. if x_i was never changed in a memory transition τ_j with $j < i$, then its value defaults to ε, and we define $\eta_i := \varepsilon$,
   b. if x_i was reset in a memory transition τ_j with $j < i$, and not changed between τ_j and τ_i, we define $\eta_i := \varepsilon$,
   c. otherwise, there exist well-defined $1 \leq j < j' \leq i$ such that x_i was opened in transition τ_j and closed in $\tau_{j'}$, and not changed between τ_j and τ_i. Hence, we define $\eta_i := v_j u_j \cdots u_{j'}$.

The constraints $\mathsf{C}_{M_{q_i,p_i}}(u_i)$ check that each u_i conforms to a sequence of transitions that takes M from q_i to p_i, without using memory transitions. The conjunction over the $i \in T$ ensures that the memory actions that are not memory recalls consume terminals correctly, and the conjunction over the $i \in R$ ensures that each memory recall refers to the right part of the consumed input. Hence, for all $w \in \Sigma^*$, $w \models \varphi_\kappa$ if and only if $w \in \mathcal{L}(M)$, and there is an accepting run of M on w that can be condensed to κ. It is easily seen that φ_κ can be constructed in polynomial time (and, hence, its size is polynomial in $|Q|$): We need to construct $\ell + 1 \leq |Q|$ automata M_{q_i,p_i}, each of which has at most $|Q|$ states and at most $|Q|^2$ transitions. Determining each η_i is also possible in time $O(|Q|)$, be checking the previous transitions τ_j.

This concludes the proof for the case of $M_i \in \mathsf{TMFA}^{\mathsf{rej}}_{\mathsf{mcf}}$. For the case where the failure state is accepting, we need to add a small extension: Instead of only considering condensed runs for runs that reach a final state, we also need to consider the runs that end in a memory recall that fails. Hence, we consider a condensed run κ of length $\ell < |Q|$, such that τ_ℓ is a memory recall transition for a variable $x \in \{1, \ldots, k\}$. Then we construct φ_κ almost as explained above. The only difference is that we replace the equation $(v_\ell = \eta_\ell)$ in the conjunction over the elements of R with the following formula:

$$\Big(\exists z\colon (\eta_\ell = v_\ell z) \wedge \mathsf{C}_A(v_\ell) \wedge \mathsf{C}_A(z)\Big) \vee \Big(\bigvee_{a\in\Sigma}\bigvee_{b\in\Sigma\setminus\{a\}} \exists y, z_1, z_2\colon (v_\ell = yaz_1) \wedge (\eta_\ell = ybz_2)\Big),$$

where A is the minimal DFA for $\mathcal{L}(A) = \Sigma^+$. The left part of this disjunctions describes all cases where v_ℓ is non-empty and a proper prefix of the content of x; the right part describes all cases where v_ℓ and the content of x differ at at least one position (recall that the content of x at this point of the run is represented by η_ℓ). Hence, this formula describes all cases where this condensed run ends in a memory recall failure.

This shows that the approach also works for $M \in \mathsf{TMFA}^{\mathsf{acc}}_{\mathsf{mcf}}$. Hence, given $M_1, \ldots, M_k \in \mathsf{TMFA}_{\mathsf{mcf}}$, we can decide whether the intersection of all $\mathcal{L}(M_i)$ is empty by guessing a condensed run κ_i for each M_i. If $M_i \in \mathsf{TMFA}^{\mathsf{rej}}_{\mathsf{mcf}}$, we only need to consider runs that end in final states; if $M_i \in \mathsf{TMFA}^{\mathsf{acc}}_{\mathsf{mcf}}$, we also need to consider runs that end in memory recall failures. Either way, the length of κ_i is bounded by the number of states in M_i. We then transform in polynomial time each κ_i into an $\mathsf{EC}^{\mathsf{reg}}$-formula φ_i, and combine these into $\varphi := \bigwedge_{i=1}^{n} \varphi_i$. Then $w \models \varphi$ if and only if $w \in \mathcal{L}(M_i)$ for all i; and as satisfiability of $\mathsf{EC}^{\mathsf{reg}}$-formulas can be decided in PSPACE, intersection emptiness is also decidable in PSPACE. As we already showed hardness at the very beginning of this proof, we conclude that the problem is PSPACE-complete. ◀

A.18 Proof of Proposition 41

A word equation is a tuple (η_L, η_R) with $\eta_L, \eta_R \in (\Sigma \cup \Xi)^*$, and a solution is a homomorphism $\sigma\colon (\Sigma \cup \Xi)^* \to \Sigma^*$ with $\sigma(a) = a$ for all $a \in \Sigma$, such that $\sigma(\eta_L) = \sigma(\eta_R)$. Even proving that it is decidable whether a word equation has a solution was by no means trivial (see Diekert [17, 18] for a detailed and a recent survey). Considering this, one might be under the impression that using this for the proof of Theorem 29 is excessive. But even intersection emptiness for $\mathsf{DRX}_{\mathsf{vsf}}$ is at least as hard as the satisfiability problem for word equations:

▶ **Proposition 41.** *Given a word equation η over Σ, we can construct in linear time $\alpha_L, \alpha_R \in \mathsf{DRX}_{\mathsf{vsf}}$ over $\Sigma \cup \{\#\}$ such that $\mathcal{L}(\alpha_L) \cap \mathcal{L}(\alpha_R) \neq \emptyset$ holds if and only if η has a solution.*

Proof. Let $\eta = (\eta_L, \eta_R)$, and assume that $x_1, \ldots, x_k$ are the variables that occur in η. Let $\#$ be a new terminal letter, $\# \notin \Sigma$, and define

$$\alpha_L := \langle x_1\colon \Sigma^*\rangle \# \langle x_2\colon \Sigma^*\rangle \# \cdots \langle x_k\colon \Sigma^*\rangle \# \beta_L,$$
$$\alpha_R := \langle x_1\colon \Sigma^*\rangle \# \langle x_2\colon \Sigma^*\rangle \# \cdots \langle x_k\colon \Sigma^*\rangle \# \beta_R,$$

where β_L and β_R are obtained from η_L and η_R (respectively) by replacing each occurrence of a variable x_i with the reference $\& x_i$. As $\# \notin \Sigma$, and as β_L and β_R consist only of a chain of terminals and variable references, $\alpha_L, \alpha_R \in \mathsf{DRX}$. Furthermore, $w \in \mathcal{L}(\alpha_L) \cap \mathcal{L}(\alpha_R)$ holds if and only if there is a homomorphism $\sigma\colon (\Sigma \cup \Xi)^* \to \Sigma^*$ with $\sigma(a) = a$ for all $a \in \Sigma$ such that

$$\begin{aligned} w &= \sigma(x_1) \# \sigma(x_2) \# \cdots \sigma(x_k) \# \sigma(\eta_L) \\ &= \sigma(x_1) \# \sigma(x_2) \# \cdots \sigma(x_k) \# \sigma(\eta_R), \end{aligned}$$

which holds if and only if there is a solution σ of η. ◀

A.19 Proof of Theorem 30

Proof. We use the simple fact that $L_1 \subseteq L_2$ holds if and only if $L_1 \cap (\Sigma^* \setminus L_2) = \emptyset$. Due to Proposition 38, we can assume that M_2 is complete (see Section A.4).

While we could use Theorem 29 together with Theorem 6 to show that the inclusion problem is decidable, the proof of Theorem 6 uses a construction that can lead to an exponential blowup in the number of states. The reason for this is that even in a complete DTMFA, we cannot simply toggle the acceptance behaviour of states, as the automaton might continue its computation by recalling memories that contain ε.

But as we shall see, it is possible to adapt the proof of Theorem 29 to handle this as well. First, note that we do not need to consider how to handle memory recall failures, as this is already part of the proof (we can simply add or remove the modifications that we discussed for $\mathsf{DTMFA}^{\mathsf{acc}}$). The first modification is that the algorithm now guesses a condensed run κ that ends in an state p_ℓ that is not accepting. But to ensure that we can treat this state as an accepting state, we need to ensure that no accepting state can be reached from it. Instead of putting this into the formula, we make an additional guess: For each variable $x \in \{1, \ldots, k\}$, we also guess a language L_x such that $L_x = \{\varepsilon\}$ or $L_x = \Sigma^+$. Formally, in addition to κ and ℓ, the algorithm guesses a function $f\colon \{1, \ldots, k\} \to \{M_\varepsilon, M_{\Sigma^+}\}$, where M_ε and M_{Σ^+} are NFA with $\mathcal{L}(M_\varepsilon) = \{\varepsilon\}$ and $\mathcal{L}(M_{\Sigma^+}) = \Sigma^+$.

It then checks whether it is possible to reach an accepting state from q, using only ε-transitions and memory recalls for variables x with $f(x) = M_\varepsilon$. If that is the case, the algorithm rejects the guess. Otherwise, it constructs a formula $\varphi_{\kappa,f}$, which is obtained from φ_κ by adding the following formula to the conjunction:

$$\exists y_1, \ldots, y_k\colon \bigwedge_{x \in \{1,\ldots,k\}} \big((y_x = \hat{\eta}_x) \land \mathsf{C}_{f(x)}(y_i)\big),$$

where each $\hat{\eta}_x$ is chosen to represent the content of the variable x in p_ℓ, like the η_j in the proof of Theorem 29. Hence, this formula checks whether the contents of the variables when reaching the state p_ℓ conform to the guessed function f.

Hence, for every word that would be rejected by M_2, we can guess appropriate ℓ, κ, and f, which allows us to decide the intersection emptiness of $\mathcal{L}(M_1)$ and $(\Sigma^* \setminus \mathcal{L}(M_2))$ in PSPACE as in the proof of Theorem 29. Hence, inclusion is decidable in PSPACE. ◀

A.20 Proof of Proposition 31

We first need a more formal definition of ℓ-determinism. Let $\ell \in \mathbb{N}$ and let $u, v \in \Sigma^*$. The words u and v are *ℓ-prefix equivalent*, denoted by $u \equiv_\ell v$, if u is a prefix of v, v is a prefix of u or their longest common prefix has a size of at least ℓ. By $u \not\equiv_\ell v$, we denote that u and v are not ℓ-prefix equivalent. Note that in order to check for two words u and v whether or not $u \equiv_k v$, it is sufficient to compare the first $\min\{k, |u|, |v|\}$ symbols of u and v.

Based on this, we define the notion of *ℓ-deterministic* TMFA as a relaxation of the criteria of DTMFA: In contrast to the latter, an ℓ-deterministic $M \in \mathsf{TMFA}(k)$ can have states q with multiple outgoing memory recall-transitions, as long as these recall distinct memories, and for every reachable configuration $(q, v, (u_1, r_1), \ldots, (u_k, r_k))$ of M, $u_i \not\equiv_\ell u_j$ holds for all $i \neq j$ that appear on the recall transitions of q.

Proof. Let $M \in \ell\text{-}\mathsf{DTMFA}(k)$. We transform M into an equivalent DTMFA M' as follows. We implement k auxiliary memories (called *state-memories* in the following) in the finite state control, which can store words of length at most ℓ, i. e., we replace every state q by states

$[q, m_1, m_2, \ldots, m_k]$, where $m_i \in \Sigma^*$, $|m_i| \leq \ell$, $1 \leq i \leq k$. The general idea is that M' simulates M in such a way that whenever M reaches a configuration $(q, v, (u_1, r_1), \ldots, (u_k, r_k))$, then M' reaches the configuration with state $[q, m_1, m_2, \ldots, m_k]$ and memory configurations (u'_i, r_i), $1 \leq i \leq k$, such that, for every i, $1 \leq i \leq k$, $u_i = m_i u'_i$ and if $u'_i \neq \varepsilon$, then $|m_i| = k$. This can be achieved as follows.

Initially, all memories and state-memories are empty and closed (to this end, the finite state control contains a flag for each state-memory, indicating whether or not it is open). If M consults memory i, then M' consumes the content of the state-memory i from the input, symbol by symbol, and then applies a memory recall instruction on memory i (note that memory i might be empty). If the consumption of the content of state-memory i fails, i. e., it is not a prefix of the remaining input, then we move to the state [trap].

Whenever M opens memory i, M' empties the state-memory i and marks it as open, but does not yet open memory i. The scanned input is now stored as follows. If a single symbol is read and the state-memory currently stores a word of length at most $\ell - 1$, then this symbol is appended to the state-memory (furthermore, if the new symbol exhausts the state-memory's capacity, then memory i is opened), and if the state-memory already stores a word of length ℓ, then the symbol is automatically stored in the open memory i.

On the other hand, if M consumes a prefix u of the input by a memory recall instruction for some memory j, i. e., in M', the state-memory j stores some u' and memory j stores some u'' with $u = u'u''$, then this is simulated by M' as follows. We start consuming u' symbol by symbol and store every symbol in state-memory i. If this is possible without exhausting the capacity of state-memory i (i. e., state-memory i now stores a word of length at most $\ell - 1$), then $|u'| < \ell$, which implies $u'' = \varepsilon$ and we are done. On the other hand, if the consumption of u' exhausts the state-memories capacity, i. e., $u' = v'v''$, where v' is the largest prefix that fits in state-memory i (note that $v' = u$ is possible), then we open memory i and fill it with $v''u''$ by first consuming v'' symbol by symbol and then consulting memory j.

We implement the modifications from above in such a way that whenever in M there is a nondeterministic choice of the form that, for some state q and several $i_1, i_2, \ldots, i_s$, $1 \leq i_j \leq k$, $1 \leq j \leq s$, each $\delta(q, i_j)$, $1 \leq j \leq s$, is defined (note that, since M is ℓ-deterministic, these are the only possible non-deterministic choices), then this is implemented in M' by s many ε-transitions from the states $[q, m_1, m_2, \ldots, m_k]$. Since the modifications from above do not require any nondeterminism, there is a one-to-one correspondence between the nondeterministic choices of M and M'. We further note that, for every j, $1 \leq j \leq s$, the ε-transition for consulting memory i_j is followed by a path of states, in which the content of state-memory i_j is consumed symbol by symbol, followed by a recall of memory i_j (and, simultaneously, for every open memory i, the state-memory is filled with the consumed symbols until it is full and then memory i is opened). The memory recall performed by this path of states either fails, which can happen in the phase where the content of the state-memory is matched with the input or in the actual recall of the memory, or it successfully simulates the memory recall. We shall now describe how the nondeterministic choices of M' can be removed.

Instead of nondeterministically choosing one of these paths, we carry them out in parallel as follows. We start consuming a prefix of the remaining input and compare it, symbol by symbol, with the contents of the state-memories i_j, $1 \leq j \leq s$. Whenever the next input symbol does not match the next symbol of a state-memory i_j, we mark this memory as *inactive* and ignore it from now on. If all memories are inactive, we change to state [trap] and if there is exactly one active memory i_j left, we conclude the consultation of this memory (i. e., we match the remaining part of the state-memory i_j with the input and then consult memory i_j). In particular, we note that if a state-memory has been completely and successfully matched with a prefix of the input and there is a another memory still active, then the contents of these memories are ℓ-prefix equivalent, which is a contradiction to the ℓ-determinism of M. Consequently, we encounter the situation that either all memories are inactive or that exactly one active one is left, before a state-memory is completely matched with a prefix of the input. Obviously, this procedure is completely deterministic and it results in an equivalent automaton. ◀

A.21 Proof of Proposition 32

Proof. Before we proceed to the actual proof, we briefly discuss why it is possible to treat non-deterministic regex as an input for the problem, considering that the number of transitions in $\mathcal{M}(\alpha)$ can be exponential (in the number of variables of α). While this is true in general, the non-deterministic regex that have these blowups are also not ℓ-deterministic: As soon as an $M \in$ TMFA has more than one transition from one state to another, it is not ℓ-deterministic. Hence, we can use an algorithm that decides ℓ-determinism for TMFA to decide ℓ-determinism for RX by converting every input $\alpha \in$ RX into $\mathcal{M}(\alpha)$ according to the proof of Theorem 17, but aborting if $G_{\tilde{\alpha}}$ contains nodes u and v with at least two edges from u and v (if these occur, α can be rejected as not ℓ-deterministic, regardless which ℓ was chosen).

Upper bounds: In order to prove the upper bounds, let $M \in$ TMFA and $\ell \geq 1$. Assume that M is not deterministic, but only violates the criteria by having states with multiple outgoing memory recall transitions for different variables (if any other violation of the criteria occurs, M cannot be ℓ-deterministic). Now, M is not ℓ-deterministic if and only if there exists a state q in M that has outgoing memory recall transitions for two different variables x and y, and there is a run of M that reaches q while x and y contain words w_x and w_y (respectively) such that $w_x \equiv_\ell w_y$. We show this property can be decided in PSPACE in general, and in NP if M is memory-cycle-free. The claim of the Proposition follows then directly if M is memory-cycle-free, and from the closure of PSPACE under complementation in the general case.

We first consider the general case: The PSPACE algorithm guesses a state q that has two outgoing memory recall transitions for variables x and y. It then guesses its way from q_0 through the automaton, while storing for each variable z of M (not just x and y) the first ℓ letters of the stored word w_z (in order to determine these for a memory z, it suffices to know all terminal edges that are traversed while z is open, and at most ℓ letters of each memory z' that is referenced while z is open). If the algorithm reaches q while $w_x \equiv_\ell w_y$, the algorithm correctly identifies M as not ℓ-deterministic. Hence, this can be decided in PSPACE.

For the memory-cycle-free case, we combine this with the condensed runs from the proof of Theorem 29. The NP-algorithm first guesses a condensed run κ of M that ends at q with outgoing memory recall transitions for x and y. In order to determine the first ℓ letters of each variable, it then guesses a prefix u_i of length at most ℓ for each transition from a q_i to a p_i, and checks whether there is a word in $\mathcal{L}(M_{q_i,p_i})$ that has u_i as a prefix (where the ε-NFA M_{q_i,p_i} is obtained as in the proof of Theorem 29: Remove all memory transitions from M, and take q_i as starting and p_i as only accepting state). It then computes the first ℓ letters of w_x and w_y, which suffice to determine whether $w_x \equiv_\ell w_y$. If $w_x \equiv_\ell w_y$, the algorithm correctly identifies M as not ℓ-deterministic. Hence, for memory-cycle-free TMFA, the absence of ℓ-determinism can be decided in NP, which means that ℓ-determinism can be decided in coNP.

Lower bound for $\mathsf{TMFA}_{\mathsf{mcf}}$ and $\mathsf{RX}_{\mathsf{vsf}}$: We prove this claim with a reduction from the 3-satisfiability problem, which is well-known to be NP complete (cf. Garey and Johnson [28]). Let φ be a formula in 3-conjunctive normal form, with variables $V = \{v_1, \ldots, v_k\}$, $k \geq 1$, where $\varphi := \bigwedge_{i=1}^{n} \varphi_i$ with $\varphi_i = (\lambda_{i,1} \vee \lambda_{i,2} \vee \lambda_{i,3})$, and $\lambda_{i,j} \in \{v, \neg v \mid v \in V\}$ for all $1 \leq i \leq n$ and $1 \leq j \leq 3$.

Our goal is to construct a $\beta \in \mathsf{RX}_{\mathsf{vsf}}$ that is not ℓ-deterministic if and only if there is an assignment to the variables in V that satisfies φ. As the latter problem is NP-complete, deciding whether a vsf-regex is ℓ-deterministic is coNP-hard. To this end, we first construct an $\alpha \in \mathsf{DRX}_{\mathsf{vsf}}$ that has a variable z that can only contain ε if φ has a satisfying assignment, and that otherwise contains a word from $\{\mathtt{a}\}^+$.

We then define $\beta := \alpha \cdot \langle z_1 \colon \mathtt{b}^\ell \rangle \langle z_2 \colon \& z\, \mathtt{b}^\ell \rangle (\& z_1 \vee \& z_2)$. Note that β is not ℓ-deterministic if and only if ε can be assigned to z; as otherwise, z always contains some word from $\mathtt{a}^+\mathtt{b}$, which means that z_1 and z_2 already differ on the first letter.

We implement this by modeling each variable $v_i \in V$ of φ with two variables x_i and $\hat{x}_i$ in α, where an assignment of 1 to v_i is modeled by setting x_i to ε and $\hat{x}_i$ to $\mathtt{a}$, while assigning 0 is modeled by setting x_i to $\mathtt{a}$ and $\hat{x}_i$ to ε. Keeping this in mind, we define $\alpha := \alpha_{\mathsf{init}} \cdot \alpha_{\mathsf{sat}}$, where

$$\alpha_{\mathsf{init}} := \alpha^1_{\mathsf{init}} \cdots \alpha^k_{\mathsf{init}},$$
$$\alpha^i_{\mathsf{init}} := \big((\mathtt{a}\langle x_i \colon \varepsilon\rangle\langle \hat{x}_i \colon \mathtt{a}\rangle) \vee (\mathtt{b}\langle x_i \colon \mathtt{a}\rangle\langle \hat{x}_i \colon \varepsilon\rangle)\big)$$

for $1 \leq i \leq k$, as well as

$$\alpha_{\mathsf{sat}} := \alpha^1_{\mathsf{sat}} \cdots \alpha^n_{\mathsf{sat}} \cdot \langle z \colon \& y_1 \cdots \& y_n \rangle,$$
$$\alpha^i_{\mathsf{sat}} := (\mathtt{a} \cdot \alpha^{i,1}_{\mathsf{lit}}) \vee \Big(\mathtt{b}\big((\mathtt{a} \cdot \alpha^{i,2}_{\mathsf{lit}}) \vee (\mathtt{b} \cdot \alpha^{i,3}_{\mathsf{lit}})\big)\Big),$$
$$\alpha^{i,j}_{\mathsf{lit}} := \begin{cases} \langle y_i \colon \& x_l \rangle & \text{if } \lambda_{i,j} = v_l, \\ \langle y_i \colon \& \hat{x}_l \rangle & \text{if } \lambda_{i,j} = \neg v_l \end{cases}$$

for $1 \leq i \leq n$, and $1 \leq j \leq 3$.

Now, observe that α is deterministic, as each part of a disjunction starts with a unique first letter ($\mathtt{a}$ or $\mathtt{b}$); and α is obviously vstar-free. To see that α can assign ε to z if and only if φ has a satisfying assignment, we read α from left to right: First, α_{init} ensures that for each pair of variables x_i and $\hat{x}_i$, exactly one is bound to ε, and the other to $\mathtt{a}$ (recall that setting x_i to ε corresponds to assigning 1 to v_i). Next, for each clause φ_i, α^i_{sat} stores the value of one of the literals $\lambda_{i,j} \in \{v_l, \neg v_l\}$ under the chosen assignment y_i, by recalling the appropriate x_l or $\hat{x}_l$. Thus, y_i can only contain ε if the assignment satisfies φ_i. Finally, all y_i are concatenated, and the result is stored in z. Hence, z can only contain ε if all clauses φ_i are satisfied, which means that φ is satisfied. Likewise, each satisfying assignment can be used to make the appropriate choices in the α^i_{init} and α^j_{sat} such that z contains ε.

Hence, as explained above, β is ℓ-deterministic if and only if φ has no satisfying assignment, which means that deciding whether a vstar-free regex is ℓ-deterministic is coNP-hard. As we already showed the matching upper bound, the problem is coNP-complete.

Lower bound for TMFA and RX: We show this with a reduction from the intersection emptiness problem for DFA, which is defined as follows: Given $M_1, \ldots, M_n \in \mathsf{DFA}$ for some $n \geq 2$, is there a $w \in \Sigma^*$ with $w \in \mathcal{L}(M_i)$ for all $1 \leq i \leq n$? This problem is PSPACE-complete (cf. Kozen [35]).

As in the case for vstar-free regex, we first construct an $\alpha \in \mathsf{DRX}$ that has a variable z such that it is possible to assign ε to z if and only if the intersection of the $\mathcal{L}(M_i)$ is not empty (and which is set to a word from $\{\mathtt{a}\}^+$ otherwise), and then define

$$\beta := \alpha \cdot \langle z_1 \colon \mathtt{b}^\ell \rangle \langle z_2 \colon \& z\, \mathtt{b}^\ell \rangle (\& z_1 \vee \& z_2).$$

Again, β is not ℓ-deterministic if and only if ε can be assigned to z; as otherwise, z always contains some word from $\mathtt{a}^+\mathtt{b}$.

Consider $M_1, \ldots, M_n \in \mathsf{DFA}$ with $M_i = (\Sigma, Q_i, q_{i,0}, \delta_i, F_i)$. In order to simplify the construction, we assume $Q_i = \{q_{i,0}, \ldots, q_{i,m}\}$ for some $m \geq 1$, and $\Sigma \supseteq \{\mathtt{a}, \mathtt{b}, \mathtt{c}_0, \ldots, \mathtt{c}_{\max(m,n)}\}$. We shall discuss in the proof how the construction can be adapted to a binary alphabet, but using an unbounded alphabet is simpler.

The main idea of the construction is that each state $q_{i,j}$ is represented by a variable $x_{i,j}$, which can take either $\mathtt{a}$ or ε as values. The regex α contains a subexpression α_{iter} which uses a Kleene star to simulate all M_i in parallel on the same input. In particular, it ensures that $x_{i,j}$ can be set to ε if and only if M_i can enter state $q_{i,j}$ at the current point of the parallel simulation. Using this, we shall see that it is possible to set z to ε if and only if all M_i can reach an accepting state at the same time. We define

$$\alpha := \alpha_{\mathsf{init}} \cdot \big(\mathtt{a} \cdot \alpha_{\mathsf{iter}}\big)^* \cdot \mathtt{b} \cdot \alpha_{\mathsf{acc}}$$

Before we define the subexpressions of α, we observe that the use of the Kleene star does not affect determinism, as the terminals $\mathtt{a}$ and $\mathtt{b}$ signal whether there should be another iteration of the star or not (respectively). The subexpressions of α are defined as follows:

$$\alpha_{\mathsf{init}} := \alpha^1_{\mathsf{init}} \cdots \alpha^n_{\mathsf{init}},$$
$$\alpha^i_{\mathsf{init}} := \langle x_{i,0} \colon \varepsilon \rangle \langle x_{i,1} \colon \mathtt{a} \rangle \cdots \langle x_{i,m} \colon \mathtt{a} \rangle$$

for all $1 \leq i \leq n$. This represents that each automaton M_i is in its starting state $q_{0,i}$. Furthermore, to simulate the behaviour of the automata M_i, we define

$$\alpha_{\mathsf{iter}} := \big((\mathtt{a} \cdot \alpha^{\mathtt{a}}_{\mathsf{step}}) \vee (\mathtt{b} \cdot \alpha^{\mathtt{b}}_{\mathsf{step}})\big) \cdot \alpha_{\mathsf{switch}},$$
$$\alpha^d_{\mathsf{step}} := \alpha^{d,1}_{\mathsf{step}} \cdots \alpha^{d,n}_{\mathsf{step}},$$
$$\alpha^{d,i}_{\mathsf{step}} := \bigvee_{0 \leq j \leq m} \Big(\mathtt{c}_j \cdot \big(\bigvee_{\substack{0 \leq l \leq m, \\ \delta_i(q_{l,i}, d) = q_{j,i}}} \mathtt{c}_l \cdot \langle \hat{x}_{i,j} \colon \& x_{i,l} \rangle \big) \cdot \alpha^{i,j}_{\mathsf{dump}}\Big),$$
$$\alpha^{i,j}_{\mathsf{dump}} := \langle \hat{x}_{i,0} \colon \mathtt{a} \rangle \cdots \langle \hat{x}_{i,j-1} \colon \mathtt{a} \rangle \langle \hat{x}_{i,j+1} \colon \mathtt{a} \rangle \cdots \langle \hat{x}_{i,m} \colon \mathtt{a} \rangle,$$
$$\alpha_{\mathsf{switch}} := \alpha^1_{\mathsf{switch}} \cdots \alpha^n_{\mathsf{switch}},$$
$$\alpha^i_{\mathsf{switch}} := \langle x_{i,0} \colon \& \hat{x}_{i,0} \rangle \cdots \langle x_{i,m} \colon \& \hat{x}_{i,m} \rangle$$

for $d \in \{\mathtt{a}, \mathtt{b}\}$, $1 \leq i \leq n$, and $0 \leq j \leq m$. Each $\alpha^{d,i}_{\mathsf{step}}$ picks a pair of states $q_{i,j}$ and $q_{i,l}$ of M_i, such that $\delta(q_{i,l}, d) = q_{i,j}$. Less formally, $q_{i,j}$ is the successor state of $q_{i,l}$ on input d. The temporary variable $\hat{x}_{i,j}$ is then set to the content of $x_{i,l}$, while all other temporary variables $\hat{x}_{i,j'}$ with $j' \neq j$ are set to $\mathtt{a}$, using $\alpha^{i,j}_{\mathsf{dump}}$.

Hence, each iteration of α^d_{step} can set $\hat{x}_{i,j}$ to ε if and only if $q_{i,j}$ is the successor state on input d for a state $q_{i,l}$ such that $x_{i,l}$ contains ε. In other words, each iteration of α_{iter} uses a subexpression α^d_{step} to simulates all M_i in parallel on the input letter $d \in \{\mathtt{a}, \mathtt{b}\}$, and α_{switch} sets each $x_{i,j}$ to the same content as its corresponding temporary variable $\hat{x}_{i,j}$.

As an aside, note that it is possible to adapt the construction to a binary terminal alphabet. To do so, one replaces the disjunctions over the terminals $\mathtt{c}_j$ and $\mathtt{c}_l$ with nested disjunctions over $\mathtt{a}$ and $\mathtt{b}$, as in the expressions α^i_{sat} in the proof for the lower bound for $\mathsf{RX}_{\mathsf{vsf}}$ above.

Regardless of the number of terminal letters, we define the remaining subexpressions as follows:

$$\alpha_{\mathsf{acc}} := \alpha^1_{\mathsf{acc}} \cdots \alpha^n_{\mathsf{acc}} \cdot \langle z\colon \&y_1 \cdots \&y_n\rangle,$$
$$\alpha^i_{\mathsf{acc}} := \bigvee_{\substack{0\leq j\leq m,\\ q_{i,j}\in F_i}} \mathtt{c}_j \cdot \langle y_i\colon \&x_{i,j}\rangle$$

for all $1 \leq i \leq n$. Again, this disjunction can be adapted to a binary terminal alphabet, as described above.

It is possible to set z to ε if and only if every y_i can be set to ε. In turn, this is possible if and only if for every M_i, there is an accepting state $q_{j,i}$ such that $x_{j,i}$ can be set to ε. As established above, α_{iter} ensures that this is only possible if these states can be reached by simulating all M_i in parallel on the same input. Hence, z can be set to ε if and only if the intersection of all $\mathcal{L}(M_i)$ is not empty.

As discussed above, α is deterministic (we discussed the use of the Kleene star above, and all branches disjunctions start with characteristic letters). Hence, β is ℓ-deterministic if and only if the intersection of the $\mathcal{L}(M_i)$ is empty. As β can obviously be constructed in polynomial time, this shows that deciding whether a regex is ℓ-deterministic is PSPACE-hard. As we already established the matching upper bound, this concludes the whole proof. ◀

Furthermore, while this definition of ℓ-determinism is only concerned with choices between different variables, it is also possible to adapt the notion of 1-determinism to include the distinction between a variable and a terminal. For example, $\langle x\colon \mathtt{a}^+\rangle\mathtt{b}(\mathtt{b} \vee \&x)^*$ is not deterministic; but as the content of x always starts with $\mathtt{a}$, such cases could be considered 1-deterministic. Propositions 31 and 32 can be directly adapted to this extended notion of 1-determinism.