Characterising REGEX Languages by Regular Languages Equipped with Factor-Referencing

Markus L. Schmid

Universität Trier, FB IV–Abteilung Informatikwissenschaften,
D-54286 Trier, Germany, MSchmid@uni-trier.de

Abstract. A (factor-)reference in a word is a special symbol that refers to another factor in the same word; a reference is dereferenced by substituting it with the referenced factor. We introduce and investigate the class ref-REG of all languages that can be obtained by taking a regular language R and then dereferencing all possible references in the words of R. We show that ref-REG coincides with the class of languages defined by regular expressions as they exist in modern programming languages like Perl, Python, Java, etc. (often called REGEX languages).

Keywords: REGEX Languages, Regular Languages, Memory Automata

1 Introduction

It is well known that most natural languages contain at least some structure that cannot be described by context-free grammars and also with respect to artificial languages, e. g., programming languages, it is often necessary to deal with structural properties that are inherently non-context-free (Floyd's proof (see [10]) that *Algol 60* is not context-free is an early example). Hence, as Dassow and Păun [8] put it, "the world seems to be non-context-free." On the other hand, the full class of context-sensitive languages, while powerful enough to model the structures appearing in natural languages and most formal languages, is often, in many regards, simply *too* much. Therefore, investigating those properties of languages that are inherently non-context-free is a classical research topic, which, in formal language theory is usually pursuit in terms of *restricted* or *regulated rewriting* (see Dassow and Păun [8]), and in computational linguistics *mildly context-sensitive* languages are investigated (see, e. g., Kallmeyer [13]).

In [9], Dassow et al. summarise the three most commonly encountered non-context-free features in formal languages as *reduplication*, leading to languages of the form $\{ww \mid w \in \Sigma^*\}$, *multiple agreements*, modelled by languages of the form $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\}$ and *crossed agreements*, as modeled by $\{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^n\mathtt{d}^m \mid n, m \geq 1\}$. In this work, we solely focus on the first such feature: reduplication.

The concept of reduplication has been mainly investigated by designing language generators that are tailored to reduplications (e. g., L systems (see Kari et al. [14] for a survey), Angluin's pattern languages [2] or H-systems by Albert and Wegner [1]) or by extending known generators accordingly (e. g., Wijngaarden grammars, macro grammars, Indian parallel grammars or deterministic iteration grammars (cf. Albert and Wegner [1] and Bordihn et al. [3] and the references therein)). A more recent approach is to extend regular expressions with some kind of copy operation (e. g., pattern expressions by Câmpeanu and Yu [6], synchronized regular expressions by Della Penna et al. [15], EH-expressions by Bordihn et al. [3]). An interesting such variant are regular expressions with backreferences (REGEX for short), which play a central role in this work. REGEX are regular expressions that contain special symbols that refer to the word that has been matched to a specific subexpression. Unlike the other mentioned language descriptors, REGEX seem to have been invented entirely on the level of software implementation, without prior theoretical formalisation (see Friedl [12] for their practical relevance). An attempt to formalise and investigate REGEX and the class of languages they describe from a theoretical point of view has been started recently (see [4,6,11,16]). This origin of REGEX from application render their theoretical investigation difficult. As pointed out by Câmpeanu and Santean in [5], "we observe implementation inconsistencies, ambiguities and a lack of standard semantics." Unfortunately, to at least some extend, these conceptional problems inevitably creep into the theoretical literature as well.

Regular expressions often serve as an user interface for specifying regular languages, since finite automata are not easily defined by human users. On the other hand, due to their capability of representing regular languages in a concise way, regular expressions are deemed inappropriate for implementations and for proving theoretical results about regular languages (e. g., closure properties or decision problems). We encounter a similar situation with respect to REGEX (which, basically, are a variant of regular expressions), i. e., their widespread implementations suggest that they are considered practically useful for specifying languages, but the theoretical investigation of the language class they describe proves to be complicated. Hence, we consider it worthwhile to develop a characterisation of this language class, which is independent from actual REGEX.

To this end, we introduce the concept of *unresolved* reduplications on the word level. In a fixed word, such a reduplication is represented by a pointer or reference to a factor of the word and resolving or dereferencing such a reference is done by replacing the pointer by the value it refers to, e. g.,

$$w = \mathtt{a}\,\underbrace{\mathtt{b}\,\mathtt{a}\,\rlap{$\overbrace{\mathtt{c}\,\mathtt{b}}^{y}$}\mathtt{c}}_{x}\,\mathtt{b}\,\mathtt{c}\,\overbrace{x\,\mathtt{c}\,\mathtt{b}}^{z}\,z\,y\,\mathtt{a}\,,$$

where the symbols x, y and z are pointers to the factors marked by the brackets labelled with x, y and z, respectively. Resolving the references x and y yields $\mathtt{abacbcbaccb}z\mathtt{cba}$ and resolving reference z leads to $\mathtt{abacbcbaccbbaccbcba}$. Such words are called ref-words and sets of ref-words are ref-languages. For a ref-word w, $\mathcal{D}(w)$ denotes the word w with all references resolved and for a ref-language L, $\mathcal{D}(L) := \{\mathcal{D}(w) \mid w \in L\}$. We shall investigate the class of ref-regular languages, i. e., the class of languages $\mathcal{D}(L)$, where L is both regular and a ref-language, and, as our main result, we show that it coincides with the class of REGEX languages. Furthermore, by a natural extension of classical finite automata, we obtain a very simple automaton model, which precisely describes the class of ref-regular languages (= REGEX languages). This automaton model is used in order to introduce a subclass of REGEX languages, that, in contrast to other recently investigated such subclasses, has a polynomial time membership problem and we investigate the closure properties of this subclass. As a side product, we obtain a very simple alternative proof for the closure of REGEX languages under intersection with regular languages; a known result, which has first been shown by Câmpeanu and Santean [5] by much more elaborate techniques.

Due to space restrictions, all formal proofs are omitted, but we give brief proof sketches for some of our results.

2 Definitions

Let $\mathbb{N} := \{1, 2, 3, \ldots\}$ and $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. For an alphabet B, the symbol B^+ denotes the set of all non-empty words over B and $B^* := B^+ \cup \{\varepsilon\}$, where ε is the empty word. For the *concatenation* of two words w_1, w_2 we write $w_1 \cdot w_2$ or simply $w_1 w_2$. We say that a word $v \in B^*$ is a *factor* of a word $w \in B^*$ if there are $u_1, u_2 \in B^*$ such that $w = u_1 v u_2$. For any word w over B, $|w|$ denotes the length of w, for any $b \in B$, by $|w|_b$ we denote the number of occurrences of b in w and for any $A \subseteq B$, we define $|w|_A := \sum_{b \in A} |w|_b$.

We use regular expressions as they are commonly defined (see, e. g., Yu [17]). By DFA and NFA, we refer to the set of deterministic and nondeterministic finite automata. Depending on the context, by DFA and NFA we also refer to an individual deterministic or nondeterministic automaton, respectively.

For any language descriptor D, by $L(D)$ we denote the language described by D and for any class $\mathfrak{D}$ of language descriptors, let $\mathcal{L}(\mathfrak{D}) := \{L(D) \mid D \in \mathfrak{D}\}$. In the whole paper, let Σ be an arbitrary finite alphabet with $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\} \subseteq \Sigma$.

2.1 References in Words, Languages and Expressions

References in Words Let $\Gamma := \{[_{x_i},]_{x_i}, x_i \mid i \in \mathbb{N}\}$, where, for every $i \in \mathbb{N}$, the pairs of symbols $[_{x_i}$ and $]_{x_i}$ are *parentheses* and the symbols x_i are *variables.* For the sake of convenience, we shall also use the symbols x, y and z to denote arbitrary variables. A *reference-word over Σ* (or *ref-word*, for short) is a word over the alphabet $(\Sigma \cup \Gamma)$. For every $i \in \mathbb{N}$, let $h_i : (\Sigma \cup \Gamma)^* \to (\Sigma \cup \Gamma)^*$ be the morphism with $h_i(z) := z$, $z \in \{[_{x_i},]_{x_i}, x_i\}$, and $h_i(z) := \varepsilon$, $z \notin \{[_{x_i},]_{x_i}, x_i\}$. A reference word is *valid* if, for every $i \in \mathbb{N}$,

$$h_i(w) = x_i^{\ell_1} \, [_{x_i} \,]_{x_i} \, x_i^{\ell_2} \, [_{x_i} \,]_{x_i} \, x_i^{\ell_3} \ldots x_i^{\ell_{k_i - 1}} \, [_{x_i} \,]_{x_i} \, x_i^{\ell_{k_i}} \,, \tag{1}$$

for some $k_i \in \mathbb{N}$ and $\ell_j \in \mathbb{N}_0$, $1 \leq j \leq k_i$. Intuitively, a reference-word w is valid if, for every $i \in \mathbb{N}$, there is a number of matching pairs of parentheses $[_{x_i}$ and $]_{x_i}$ that are not nested and, furthermore, no occurrence of x_i is enclosed by such a matching pair of parentheses. However, it is not required that w is a well-formed parenthesised expression with respect to *all* occurring parentheses.

The set of valid reference-words is denoted by $\Sigma^{[*]}$. A factor $[_x \, u \,]_x$ of a $w \in \Sigma^{[*]}$ where the occurrences of $[_x$ and $]_x$ are matching parentheses is called a *reference for variable* x, and u is the *value* of this reference. A reference is a *first order reference*, if its value does not contain another reference and it is called *pure*, if it is a first order reference and its value does not contain variables. Two references of some ref-word w are *overlapping* if one reference contains exactly one of the delimiting parentheses of the other reference, e. g., in $w_1[_x w_2[_y w_3]_x w_4]_y w_5$ the references $[_x w_2[_y w_3]_x$ and $[_y w_3]_x w_4]_y$ are overlapping. Let $w \in \Sigma^{[*]}$ and let x be a variable that occurs in w. An occurrence of a variable x in w that is not preceded by a reference for x is called *undefined.* Every occurrence of a variable x in w that is not undefined *refers* to the reference for x, which precedes this occurrence. This definition is illustrated by Equation 1, where all $k_i - 1$ references for variable x_i are shown and, for every j, $1 \leq j \leq k_i - 1$, the ℓ_{j+1} occurrences of x_i between the j^{th} and $(j+1)^{\text{th}}$ reference for x_i are exactly the occurrences of x_i that refer to the j^{th} reference for variable x_i.

We consider the following examples:

$$\begin{aligned}
w_1 &:= [_x\mathtt{ab}]_x[_y\mathtt{c}x\mathtt{b}]_y[_x\mathtt{bb}y]_x\mathtt{c}y\mathtt{b}y[_y x]_y\mathtt{cc}, \quad & w_2 &:= [_x\mathtt{ba}x]_x\mathtt{a}x[_x\mathtt{bc}]_x[_x\mathtt{ba}[_x\mathtt{a}]_x\mathtt{a}]_x x\,,\\
w_3 &:= [_x[_y\mathtt{b}]_x\mathtt{c}x[_x\mathtt{b}]_y xzy\mathtt{b}[_y\mathtt{c}z]_y z[_z\mathtt{cc}]_x]_z, \quad & w_4 &:= [_x\mathtt{a}[_y\mathtt{b}[_z\mathtt{bba}]_z\mathtt{c}]_y\mathtt{b}y\mathtt{b}]_x xy\,.
\end{aligned}$$

The words w_1, w_3 and w_4 are valid ref-words, whereas w_2 is not valid. Moreover, all references of w_1 are first order references, the reference for variable x in w_4 is not a first order reference and the first reference in w_3 is pure. The word w_3 contains an undefined occurrence of a variable and overlapping references. For the sake of convenience, from now on, we call valid ref-words simply ref-words. If a word over $(\Sigma \cup \Gamma)$ is not a ref-word, then we always explicitly state this.

Next, we define how a ref-word over Σ can be dereferenced, i. e., how it can be transformed into a (normal) word over Σ. To this end, let $w \in \Sigma^{[*]}$. The *dereference* of w, denoted by $\mathcal{D}(w)$, is constructed by first deleting all undefined occurrences of variables in w and then substituting all pure references and their variables by its value (ignoring possible parentheses in the value), until there is no pure reference left. A formal proof that $\mathcal{D}(w)$ is well-defined and in fact a word over Σ is straightforward and left to the reader. Next, we illustrate this definition with an example:

$$\begin{aligned}
&\mathcal{D}(z\mathtt{a}[_z x[_x y\mathtt{b}[_y\mathtt{c}]_x\mathtt{b}x[_x\mathtt{c}]_y\mathtt{b}]_x y\mathtt{c}]_z x\mathtt{c}z) = \mathcal{D}(\mathtt{a}[_z[_x\mathtt{b}[_y\mathtt{c}]_x\mathtt{b}x[_x\mathtt{c}]_y\mathtt{b}]_x y\mathtt{c}]_z x\mathtt{c}z) =\\
&\mathcal{D}(\mathtt{a}[_z\underset{x}{\underline{\mathtt{b}[_y\mathtt{c}}}\,\mathtt{b}\,\underset{x}{\underline{\mathtt{bc}}}[_x\mathtt{c}]_y\mathtt{b}]_x y\mathtt{c}]_z x\mathtt{c}z) = \mathcal{D}(\mathtt{a}[_z\mathtt{b}\,\underset{y}{\underline{\mathtt{cbbc}[_x\mathtt{c}}}\,\mathtt{b}]_x\,\underset{y}{\underline{\mathtt{cbbcc}}}\,\mathtt{c}]_z x\mathtt{c}z) =\\
&\mathcal{D}(\mathtt{a}[_z\mathtt{bcbbc}\,\underset{x}{\underline{\mathtt{cb}}}\,\mathtt{cbbccc}]_z\,\underset{x}{\underline{\mathtt{cb}}}\,\mathtt{c}z) = \mathtt{a}\,\underset{z}{\underline{\mathtt{bcbbccbcbbccc}}}\,\mathtt{cbc}\,\underset{z}{\underline{\mathtt{bcbbccbcbbccc}}}\;.
\end{aligned}$$

We point out that ref-words are similar to Lempel-Ziv compression. However, here we are exclusively concerned with language theoretic aspects of ref-words.

References in Languages For every $i \in \mathbb{N}$, let $\Gamma_i := \{[_{x_j},]_{x_j}, x_j \mid j \leq i\}$. A set of ref-words L is a *ref-language* if $L \subseteq (\Sigma \cup \Gamma_i)^*$, for some $i \in \mathbb{N}$. For the sake of convenience, we simply write $L \subseteq \Sigma^{[*]}$ to denote that L is a ref-language. For every ref-language L, we define the *dereference of* L by $\mathcal{D}(L) := \{\mathcal{D}(w) \mid w \in L\}$ and, for any class $\mathfrak{L}$ of ref-languages, $\mathcal{D}(\mathfrak{L}) := \{\mathcal{D}(L) \mid L \in \mathfrak{L}\}$.

An $L \subseteq \Sigma^{[*]}$ is a *regular ref-language* if L is regular. A language L is called *ref-regular* if it is the dereference of a regular ref-language, i. e., $L = \mathcal{D}(L')$ for some regular ref-language L'. For example, the copy language $L_c := \{ww \mid w \in \Sigma^*\}$ is ref-regular, since $L_c = \mathcal{D}(L'_c)$, where L'_c is the regular ref-language $\{[_x\, w\,]_x\, x \mid w \in \Sigma^*\}$. The class of ref-regular languages is denoted by ref-REG.

By definition, REG $\subseteq$ ref-REG and it can be easily shown that ref-REG is contained in the class of context-sensitive language. On the other hand, the class of context-free languages is not included in ref-REG, e. g., $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\} \notin$ ref-REG (see Câmpeanu et al. [4]). Other interesting examples of ref-regular languages are the set of imprimitive words: $\mathcal{D}(\{[_x\, w\,]_x\, x^n \mid w \in \Sigma^*, n \geq 1\})$, the set of words $\mathtt{a}^n$, where n is not prime: $\mathcal{D}(\{[_x\, \mathtt{a}^m\,]_x\, x^n \mid m, n \geq 2\})$ and the set of bordered words: $\mathcal{D}(\{[_x\, u\,]_x\, v\, x \mid u, v \in \Sigma^*, |u| \geq 1\})$.

References in Expressions If we use the concept of references directly in regular expressions, i. e., we use variables x in the expression and enclose subexpressions by parentheses $[_x$ and $]_x$, then we obtain *extended regular expressions with backreferences* (or REGEX for short). For more detailed definitions and further information on REGEX, we refer to [4, 5, 11, 16].

A convenient definition of the semantics of a REGEX can also be given in terms of classical regular expressions and ref-words. We can interpret a REGEX r as a classical regular expression r' over the alphabet $(\Sigma \cup \Gamma_k)$, where k is the number of backreferences in r. Now $L(r')$ is a ref-language and $\mathcal{D}(L(r'))$ is the REGEX language described by the REGEX r. This observation yields the following result.

Proposition 1. $\mathcal{L}$(REGEX) $\subseteq$ ref-REG.

On the other hand, a regular expression s with $L(s) \subseteq \Sigma^{[*]}$ does not translate into a REGEX in an obvious way, which is due to the fact that in s it is not necessarily the case that every occurrence of $[_x$ matches with an occurrence of $]_x$ and, furthermore, even matching pairs of parentheses do not necessarily enclose subexpressions. For example, the regular expression

$$s := [_{x_1}(([_{x_2}\mathtt{b}^*]_{x_1}\mathtt{a}^* x_1 [_{x_1}) + ([_{x_2}\mathtt{a}^*\mathtt{c}^*))\,\mathtt{ba}]_{x_2}(x_2 + \mathtt{d})^*]_{x_1}\mathtt{a}x_1$$

describes a ref-language, but it cannot be interpreted as a REGEX.

In the following, we say that a regular expression r over the alphabet $(\Sigma \cup \Gamma_k)$ has the REGEX *property* if it is also a valid REGEX.

3 Memories in Automata

By a natural extension of classical finite automata, we now define memory automata, which are the main technical tool for proving the results of this paper.

A memory automaton is a classical NFA that is equipped with a finite number of k memory cells, each capable of storing a word. Each memory is either *closed*, which means that it is not affected in a transition, or *open*, which means that it records the currently scanned input symbol. In a transition it is possible to *consult* a closed memory, which means that its content, if it is a prefix of the remaining input, is consumed in one step from the input and, furthermore, also stored in all the open memories. A closed memory can be opened again, but then it completely loses its previous content; thus, memories always store factors of the input. We shall now formally define the model of memory automata.

Definition 1. *For every* $k \in \mathbb{N}$*, a* k-memory automaton*, denoted by* $\mathrm{MFA}(k)$*, is a tuple* $M := (Q, \Sigma, \delta, q_0, F)$*, where* Q *is a finite set of* states*,* Σ *is a finite* alphabet*,* q_0 *is the* initial state*,* F *is the set of* final states *and*

$$\delta : Q \times (\Sigma \cup \{\varepsilon\} \cup \{1, 2, \ldots, k\}) \to \mathcal{P}(Q \times \{\mathtt{o}, \mathtt{c}, \diamond\}^k)$$

is the transition function *(where* $\mathcal{P}(A)$ *denotes the power set of a set* A*). The elements* $\mathtt{o}$*,* $\mathtt{c}$ *and* $\diamond$ *are called* memory instructions*.*

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*, for every* i*,* $1 \leq i \leq k$*,* $u_i \in \Sigma^*$ *is the* content of memory i *and* $r_i \in \{\mathtt{O}, \mathtt{C}\}$ *is the* status of memory i*. For a memory status* $r \in \{\mathtt{O}, \mathtt{C}\}$ *and a memory instruction* $s \in \{\mathtt{o}, \mathtt{c}, \diamond\}$*, we define* $r \oplus s = \mathtt{O}$ *if* $s = \mathtt{o}$*,* $r \oplus s = \mathtt{C}$ *if* $s = \mathtt{c}$ *and* $r \oplus s = r$ *if* $s = \diamond$*. Furthermore, for a tuple* $(r_1, \ldots, r_k) \in \{\mathtt{O}, \mathtt{C}\}^k$ *of memory statuses and a tuple* $(s_1, \ldots, s_k) \in \{\mathtt{o}, \mathtt{c}, \diamond\}^k$ *of memory instructions, we define* $(s_1, \ldots, s_k) \oplus (r_1, \ldots, r_k) = (s_1 \oplus r_1, \ldots, s_k \oplus r_k)$*.*

M *can change from a configuration* $c := (q, w, (u_1, r_1), \ldots, (u_k, r_k))$ *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* $\delta(q, b) \ni (p, s_1, \ldots, s_k)$ *such that* $w = v\,w'$*, where* $v = b$ *if* $b \in (\Sigma \cup \{\varepsilon\})$ *and* $v = u_b$ *and* $r_b = \mathtt{C}$ *if* $b \in \{1, 2, \ldots, k\}$*. Furthermore,* $(r'_1, \ldots, r'_k) = (r_1, \ldots, r_k) \oplus (s_1, \ldots, s_k)$ *and, for every* i*,* $1 \leq i \leq k$*,* $u'_i = u_i v$ *if* $r'_i = r_i = \mathtt{O}$*,* $u'_i = v$ *if* $r'_i = \mathtt{O}$ *and* $r_i = \mathtt{C}$*,* $u'_i = u_i$ *if* $r'_i = \mathtt{C}$*. The symbol* $\vdash^*_M$ *denotes the reflexive and transitive closure of* $\vdash_M$*. For any* $w \in \Sigma^*$*, a configuration* $(p, v, (u_1, r_1), \ldots, (u_k, r_k))$ *is* reachable *(*on input w*), if*

$$(q_0, w, (\varepsilon, \mathtt{C}), \ldots, (\varepsilon, \mathtt{C})) \vdash^*_M (p, v, (u_1, r_1), \ldots, (u_k, r_k)) \,.$$

A $w \in \Sigma^*$ *is accepted by* M *if a configuration* $(q_f, \varepsilon, (u_1, r_1), \ldots, (u_k, r_k))$ *with* $q_f \in F$ *is reachable on input* w *and* $L(M)$ *is the set of words accepted by* M*.*

For any $k \in \mathbb{N}$, $\mathrm{MFA}(k)$ is the class of k-memory automata and $\mathrm{MFA} := \bigcup_{k \geq 0} \mathrm{MFA}(k)$. We also use $\mathrm{MFA}(k)$ and MFA in order to denote an instance of a k-memory automaton or a memory automaton with some number of memories. The set $\mathcal{L}(\mathrm{MFA}) := \{L(M) \mid M \in \mathrm{MFA}\}$ is the *class of* MFA *languages.*

As an example, we observe that an MFA(3) can accept $L := \{v_1 v_2 v_3 v_1 v_2 v_2 v_3 \mid v_1, v_2, v_3 \in \{\mathtt{a}, \mathtt{b}\}^*\}$ by reading a prefix $u = v_1 v_2 v_3$ of the input and storing v_1, v_2 and v_3 in the memories 1, 2 and 3. The length of u as well as the factorisation $u = v_1 v_2 v_3$ is nondeterministically guessed. Now we can check whether the remaining input equals $v_1 v_2 v_2 v_3$ by consulting the memories. Alternatively, while reading the prefix $u = v_1 v_2 v_3$, we can also store $v_1 v_2$ and $v_2 v_3$ in only two memories, which is sufficient to check whether the remaining input equals $v_1 v_2 v_2 v_3$. We point out that this alternative, which needs one memory less, requires the factor v_2 of the input to be simultaneously recorded by both memories or, in other words, the factors recorded by the memories overlap in the input word.

Determinism in Memory Automata Let $M := (Q, \Sigma, \delta, q_0, F)$ be a k-memory automaton. M is *pseudo deterministic* if, for every $q \in Q$ and $b \in (\Sigma \cup \{\varepsilon\} \cup \{1, 2, \ldots, k\})$, $|\delta(q, b)| \leq 1$. We call a memory automaton with this property pseudo deterministic, since it is still possible that for a $q \in Q$ and $b \in (\Sigma \cup \{\varepsilon\})$, $|\bigcup_{i=1}^{k} \delta(q, i)| + |\delta(q, b)| \geq 1$.

Any MFA can be transformed into an equivalent pseudo-deterministic one by applying a variant of the subset construction that also takes the memories into account, i. e., instead of $\mathcal{P}(Q)$ as the set of new states, we use $\mathcal{P}(Q) \times \{\mathtt{O}, \mathtt{C}\}^k$.

Lemma 1. *Let $k \in \mathbb{N}$. For every $M \in \mathrm{MFA}(k)$, there exists a pseudo deterministic $M' \in \mathrm{MFA}(k)$ with $L(M) = L(M')$.*

Analogously to the definition of determinism for classical finite automata, we can define determinism for memory automata as the situation that in every state there is at most one applicable transition. More precisely, a k-memory automaton $M := (Q, \Sigma, \delta, q_0, F)$ is *deterministic* if it is ε-free and, for every $q \in Q$ and $b \in \Sigma$, $|\bigcup_{i=1}^{k} \delta(q, i)| + |\delta(q, b)| \leq 1$. We denote the class of deterministic k-memory automata by $\mathrm{DMFA}(k)$ and $\mathrm{DMFA} := \bigcup_{k \in \mathbb{N}} \mathrm{MFA}(k)$. The class $\mathcal{L}(\mathrm{DMFA})$ shall be investigated in more detail in Section 5.

Normal Forms of Memory Automata Intuitively speaking, a memory automaton is in normal form if every transition can either read a part of the input without changing the status of any memory or it changes the status of exactly one memory, but then it does not touch the input. Furthermore, in every accepting configuration, the automaton does not try to open or close memories that are already opened or closed, respectively.

Definition 2. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be an $\mathrm{MFA}(k)$, $k \in \mathbb{N}$. We say that M is in* normal form *if the following conditions are satisfied. For every transition $\delta(q, b) \ni (p, s_1, \ldots, s_k)$, if $s_i \neq \diamond$ for some i, $1 \leq i \leq k$, then $b = \varepsilon$ and $s_j = \diamond$, for all j with $1 \leq j \leq k$ and $i \neq j$. If M reaches configuration $(q, w, (u_1, r_1), \ldots, (u_k, r_k))$ in a computation and transition $\delta(q, b) \ni (p, s_1, \ldots, s_k)$ is applied next, then, for every i, $1 \leq i \leq k$, if $r_i = \mathtt{O}$, then $s_i \neq \mathtt{o}$ and if $r_i = \mathtt{C}$, then $s_i \neq \mathtt{c}$.*

MFA can be transformed into normal form, by replacing transitions that change the status of more than one memory by several transitions that satisfy the conditions of the normal form. Furthermore, in the states we can store which memories are currently open and use this information to remove transitions that open or close memories that are already open or closed, respectively.

Lemma 2. *Let $k \in \mathbb{N}$. For every $M \in \mathrm{MFA}(k)$ there exists an $M' \in \mathrm{MFA}(k)$ with $L(M) = L(M')$ and M' is in normal form.*

Above, after Definition 1, we consider an example of an MFA that uses its memories in an overlapping way. We shall now formally define this situation. Let M be a k-memory automaton, let C be a computation of M with n steps and let $1 \leq i, j \leq k$ with $i \neq j$. We say that there is an *i-j-overlap* in C if there are p, q, r, s, $1 \leq p < q < r < s < n$, such that memory i is opened in step p and closed again in step r of C and memory j is opened in step q and closed again in step s of C. A computation C is said to be *nested* if, for every i, j, $1 \leq i \leq j \leq k$, $i \neq j$, there is no i-j-overlap in C and an MFA M is *nested* if every possible computation of M is nested.

For transforming MFA into REGEX, it is crucial to get rid of these overlaps, which is generally possible. A formal definition of the construction is omitted; here, we only give the general idea. We first modify M in such a way that in the finite state control, for every currently open memory i, we store the set A_i of currently open memories that have been opened after memory i. If memory i is closed, then, for every $j \in A_i$, an i-j-overlap occurs. Thus, we close all memories $j \in A_i$ together with memory i and then, for every $j \in A_i$, we open a new auxiliary memory instead. By doing this every time a memory is closed, we make sure that no more overlaps occur. Whenever the original MFA consults a memory, then we have to consult the right auxiliary memories in the right order instead. This strategy is only applicable because, for every original memory i, at most $k - 1$ auxiliary memories are needed.

Lemma 3. *Let $k \in \mathbb{N}$. For every $M \in \mathrm{MFA}(k)$ there exists a nested $M' \in \mathrm{MFA}(k^2)$ with $L(M) = L(M')$.*

We conclude this section by pointing out that every MFA can be transformed into an equivalent one that is pseudo-deterministic, in normal form and nested.

4 Equivalence of ref-REG, $\mathcal{L}$(MFA) and $\mathcal{L}$(REGEX)

In this section, we show that the ref-regular languages, the MFA languages and the REGEX languages are identical. To this end, we shall first prove the equality $\mathcal{L}(\mathrm{MFA}) = \text{ref-REG}$ and then the inclusion $\mathcal{L}(\mathrm{MFA}) \subseteq \mathcal{L}(\mathrm{REGEX})$, which, together with Proposition 1, implies our main result:

Theorem 1. ref-REG $= \mathcal{L}(\mathrm{MFA}) = \mathcal{L}(\mathrm{REGEX})$.

Intuitively speaking, $\mathcal{L}(\mathrm{MFA}) = \text{ref-REG}$ follows from the fact that an NFA that accepts a regular ref-language can be translated into an MFA that accepts its dereference and any MFA in normal form can be translated into an NFA that accepts a regular ref-language, the dereference of which equals the language accepted by the MFA.

We recall that every transition $\delta(q, b) \ni (p, s_1, s_2, \ldots, s_k)$ of an MFA(k) in normal form is of one of the following four types:

Σ-transition: $b \in \Sigma$ and $s_i = \diamond$, $1 \leq i \leq k$.
o_i-transition: $b = \varepsilon$, $s_i = \mathsf{o}$ and, for every j, $1 \leq j \leq k$, $i \neq j$, $s_j = \diamond$.

$\mathtt{c}_i$**-transition:** $b = \varepsilon$, $s_i = \mathtt{c}$ and, for every j, $1 \leq j \leq k$, $i \neq j$, $s_j = \diamond$.
m_i**-transition:** $b \in \{1, 2, \ldots, k\}$ and $s_i = \diamond$, $1 \leq i \leq k$.

Let $\text{NFA}_{\text{ref}} := \{M \mid M \in \text{NFA}, L(M) \subseteq \Sigma^{[*]}\}$ and $\text{MFA}_{\text{nf}} := \{M \mid M \in \text{MFA}, M \text{ is in normal form}\}$. We define a mapping $\psi_{\mathcal{D}} : \text{NFA}_{\text{ref}} \to \text{MFA}_{\text{nf}}$. To this end, let $M := (Q, \Sigma \cup \Gamma_k, \delta, q_0, F) \in \text{NFA}_{\text{ref}}$. We define a k-memory automaton $\psi_{\mathcal{D}}(M) := (Q, \Sigma, \delta', q_0, F)$, where δ' is defined as follows. For every transition $\delta(q, b) \ni p$ of M, we add a transition $\delta'(q, b) \ni (p, s_1, s_2, \ldots, s_k)$ to $\psi_{\mathcal{D}}(M)$, where this transition is an Σ-transition if $b \in \Sigma$, an $\mathtt{o}_i$-transition if $b = [_{x_i}$, a $\mathtt{c}_i$-transition if $b =]_{x_i}$ and an m_i-transition if $b = x_i$. This concludes the definition of $\psi_{\mathcal{D}}(M)$ and it can be easily seen that $\psi_{\mathcal{D}}(M) \in \text{MFA}_{\text{nf}}$. Without loss of generality, we can assume that all elements of NFA_{ref} are such that the input alphabet does not contain symbols that do not occur in the accepted language. This implies that, for any two $M_1, M_2 \in \text{NFA}_{\text{ref}}$ with $M_1 \neq M_2$, $\psi_{\mathcal{D}}(M_1) \neq \psi_{\mathcal{D}}(M_2)$ is implied, which means that $\psi_{\mathcal{D}}$ is injective. Furthermore, since every transition of some $\text{MFA}(k)$ in normal form is of one of the four types described above, we can conclude that the reverse of $\psi_{\mathcal{D}}$ is an injective mapping $\psi_{\mathcal{D}}^{-1} : \text{MFA}_{\text{nf}} \to \text{NFA}_{\text{ref}}$, which implies that $\psi_{\mathcal{D}}$ is a bijection. The following lemma directly implies ref-REG $= \mathcal{L}(\text{MFA})$.

Lemma 4. *Let $M \in \text{NFA}$ with $L(M) \subseteq \Sigma^{[*]}$ and let $N \in \text{MFA}$ be in normal form. Then $\mathcal{D}(L(M)) = L(\psi_{\mathcal{D}}(M))$ and $L(N) = \mathcal{D}(L(\psi_{\mathcal{D}}^{-1}(N)))$.*

In order to conclude the proof of Theorem 1, it only remains to show that every MFA language can be expressed as a REGEX language. To do this, we use the fact that every MFA language can be described by a nested MFA (see Lemma 3). Next, we observe an obvious, but important property of nested MFA:

Proposition 2. *Let M be a nested $\text{MFA}(k)$ in normal form. There is no word $w \in L(\psi_{\mathcal{D}}^{-1}(M))$ with overlapping references.*

Let M be a fixed nested $\text{MFA}(k)$ in normal form and let $N := \psi_{\mathcal{D}}^{-1}(M)$ with transition function δ. Without loss of generality, we can assume that every transition of the form $\delta(p, b) \ni q$ with $b \in \{[_{x_i},]_{x_i} \mid 1 \leq i \leq k\}$ is such that at least one accepting state is reachable from q and, furthermore, that every state of N is reachable from the initial state. For every i, $1 \leq i \leq k$, let $n_i, m_i \in \mathbb{N}$ be such that $\delta(p_{i,j}, [_{x_i}) \ni q_{i,j}$, $1 \leq j \leq n_i$, are exactly the transitions of N labeled with $[_{x_i}$ and $\delta(r_{i,\ell},]_{x_i}) \ni s_{i,\ell}$, $1 \leq \ell \leq m_i$, are exactly the transitions of N labeled with $]_{x_i}$. For every i, j, ℓ, $1 \leq i \leq k$, $1 \leq j \leq n_i$, $1 \leq \ell \leq m_i$, let $R_{i,j,\ell}$ be the set of words that can take N from $q_{i,j}$ to $r_{i,\ell}$ without reading any occurrence of $]_{x_i}$. If some $R_{i,j,\ell}$ contains a word that is not a ref-word, then, since an accepting state can be reached from $s_{i,\ell}$, N accepts a word that is not a ref-word or a ref-word with overlapping references, which is a contradiction:

Lemma 5. *For every i, j, ℓ, $1 \leq i \leq k$, $1 \leq j \leq n_i$, $1 \leq \ell \leq m_i$, $R_{i,j,\ell} \subseteq \Sigma^{[*]}$.*

In the following, we transform N into a regular expression r and, since we want r to have the REGEX property, this has to be done in such a way that in r

every $[_{x_i}$ matches a $]_{x_i}$ and such matching parentheses enclose a subexpression. The idea to achieve this is that, for each pair of transitions $\delta(p_{i,j}, [_{x_i}) \ni q_{i,j}$ and $\delta(r_{i,\ell},]_{x_i}) \ni s_{i,\ell}$, we transform the set of words that take N from $q_{i,j}$ to $r_{i,\ell}$, i. e., the set $R_{i,j,\ell}$, into a regular expression individually. For the correctness of this construction, it is crucial that M is nested and that we transform the $R_{i,j,\ell}$ into regular expressions in a specific order, which is defined next.

Let the binary relation $\prec$ over the set $\Phi := \{(i, j, \ell) \mid 1 \leq i \leq k, 1 \leq j \leq n_i, 1 \leq \ell \leq m_i\}$ be defined as follows. For every $(i_1, j_1, \ell_1), (i_2, j_2, \ell_2) \in \Phi$, we define $(i_1, j_1, \ell_1) \prec (i_2, j_2, \ell_2)$ if and only if there is a computation of N that starts in p_{i_2,j_2} and reaches s_{i_2,ℓ_2} and takes the transitions (1) $\delta(p_{i_2,j_2}, [_{x_{i_2}}) \ni q_{i_2,j_2}$, (2) $\delta(p_{i_1,j_1}, [_{x_{i_1}}) \ni q_{i_1,j_1}$, (3) $\delta(r_{i_1,\ell_1},]_{x_{i_1}}) \ni s_{i_1,\ell_1}$ and (4) $\delta(r_{i_2,\ell_2},]_{x_{i_2}}) \ni s_{i_2,\ell_2}$ in exactly this order and no $]_{x_{i_2}}$ is read between performing transitions 1 and 4 and no $]_{x_{i_1}}$ is read between performing transitions 2 and 3.

Lemma 6. *If $(i_1, j_1, \ell_1) \prec (i_2, j_2, \ell_2)$, then $i_1 \neq i_2$. The relation $\prec$ is irreflexive, transitive and antisymmetric.*

Next, we define a procedure that turns N into a regular expression that is based on the well-known state elimination technique (see Yu [17] for details). To this end, we need the concept of an *extended finite automaton*, which is an NFA whose transitions can be labeled by regular expressions over the input alphabet.

For every $(i, j, \ell) \in \Phi$, we define $\Delta(i, j, l) := \{(i', j', \ell') \mid (i', j', \ell') \prec (i, j, \ell)\}$ and $\Phi' := \Phi$. We iterate the following steps as long as Φ' is non-empty.

Step 1 For some $(i, j, \ell) \in \Phi'$ with $|\Delta(i, j, l)| = 0$, we obtain an automaton $K_{i,j,\ell}$ from (the current version of) N by deleting all transitions $\delta(r_{i,\ell'},]_{x_i}) \ni s_{i,\ell'}$, $1 \leq \ell' \leq m_i$, and by defining $q_{i,j}$ to be the initial state and $r_{i,\ell}$ the only accepting state. Then, we transform $K_{i,j,\ell}$ into a regular expression $t_{i,j,\ell}$ by applying the state elimination technique.
Step 2 For every $(i', j', \ell') \in \Phi$, we delete (i, j, ℓ) from $\Delta(i', j', \ell')$.
Step 3 We add the transition $\delta(p_{i,j}, [_{x_i} t_{i,j,\ell}]_{x_i}) \ni s_{i,\ell}$ to N.
Step 4 We delete (i, j, l) from Φ'.
Step 5 If, for every ℓ', $1 \leq \ell' \leq m_i$, $(i, j, \ell') \notin \Phi'$, then we delete the transition $\delta(p_{i,j}, [_{x_i}) \ni q_{i,j}$ from N.
Step 6 If, for every j', $1 \leq j' \leq n_i$, $(i, j', \ell) \notin \Phi'$, then we delete the transition $\delta(r_{i,j},]_{x_i}) \ni s_{i,\ell}$ from N.

In order to see that this procedure is well-defined, we observe that as long as $\Phi' \neq \emptyset$, there is at least one element $(i, j, \ell) \in \Phi'$ with $|\Delta(i, j, l)| = 0$, which follows directly from the transitivity and antisymmetry of $\prec$ (see Lemma 6). Furthermore, from the definition of the automata $K_{i,j,\ell}$ constructed in Step 1, it can be easily verified that $L(t_{i,j,\ell}) = R_{i,j,\ell}$ holds.

The automaton obtained by this procedure, denoted by N', does not contain any transitions labeled with symbols from $\{[_{x_i},]_{x_i} \mid 1 \leq i \leq k\}$. We can now transform N' into a regular expression r by the state elimination technique. The next lemma concludes the proof of Theorem 1.

Lemma 7. *The regular expression r has the* REGEX*-property and $L(r) = L(N)$.*

5 DMFA Languages

We now take a closer look at the class of languages accepted by DMFA. As an example, we consider $\{w\mathtt{c}w \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$, which can be accepted by a DMFA(1). However, $L_{\text{copy}} := \{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\} \notin \mathcal{L}(\text{DMFA})$, for which we give a proof sketch. Let $M \in \text{DMFA}$ with $L(M) = L_{\text{copy}}$. Since $L_{\text{copy}} \notin \text{REG}$ there is a $w \in L_{\text{copy}}$, such that M first reads a prefix v of w and then consults a memory i that stores a non-empty word u. However, since M is deterministic, this means that it cannot accept any word with prefix vb, where b does not equal the first symbol of u, which is a contradiction.

Theorem 2. $\mathcal{L}(\text{DMFA}) \subset \mathcal{L}(\text{MFA})$.

Next, we note that the membership problem of this language class can be solved in linear time by simply running the DMFA.

Theorem 3. *For a given* DMFA M *and a word* $w \in \Sigma^*$, *we can decide in time* $\text{O}(|w|)$ *whether or not* $w \in L(M)$.

This contrasts the situation that for other prominent subclasses of REGEX languages (e. g., the ones investigated in [1,3,6,16]) the membership problem is usually NP-complete.[1]

REGEX languages are closed under union, but not under intersection or complementation [4,7]. For the subclass $\mathcal{L}(\text{DMFA})$, we observe a different situation:

Theorem 4. $\mathcal{L}(\text{DMFA})$ *is closed under complementation, but it is not closed under union or intersection.*

For the non-closure under intersection, we can apply the example used by Carle and Narendran [7] to prove the non-closure of REGEX languages under intersection. The closure under complementation follows from the fact that interchanging the accepting and non-accepting states of a DMFA yields a DMFA that accepts its complement. Finally, the non-closure under union follows from $\{\mathtt{a}^n\mathtt{b}\mathtt{a}^n \mid n \in \mathbb{N}\} \cup \{\mathtt{a}^m\mathtt{b}\mathtt{a}^n\mathtt{b}\mathtt{a}^k \mid m, n, k \in \mathbb{N}_0\} \notin \mathcal{L}(\text{DMFA})$, which can be shown by a similar argument used to prove that L_{copy} is not a DMFA language.[2]

In [4], which marks the beginning of the formal investigation of REGEX, Câmpeanu et al. ask whether REGEX languages are closed under intersection with regular languages, which has been answered in the positive by Câmpeanu and Santean in [5]. We can give an analogue with respect to DMFA languages:

Theorem 5. $\mathcal{L}(\text{DMFA})$ *is closed under intersection with regular languages.*

Theorem 5 follows from the fact that we can simulate a DMFA(k) M and a DFA N in parallel by a DMFA M'. The difficulty that we encounter is that if N is currently in a state q and M consumes the content u_i of a memory i from the input, then we do not know in which state N needs to change. However, earlier

[1] Other exceptions are REGEX with a bounded number of backreferences (see [16]).

[2] This also proves non-closure of DMFA languages under union with regular languages.

in this computation, memory i is filled with content u_i and at the same time we can determine and store the state in which N needs to change if u_i is consumed from the input. Since at this time we do not yet know that q will be the current state of N when memory i is consulted, for *every* state of N, we have to store which state is reached by reading u_i. In order to update these informations we use the transition function of N (when single symbols are read) and these stored informations (when memories are consulted).

This also constitutes a much simpler alternative proof for the closure of REGEX languages under intersection with regular languages, which demonstrates that MFA are a convenient tool to handle REGEX languages.

References

1. J. Albert and L. Wegner. Languages with homomorphic replacements. *Theoretical Computer Science*, 16:291–305, 1981.
2. D. Angluin. Finding patterns common to a set of strings. In *Proc. 11th Annual ACM Symposium on Theory of Computing*, pages 130–141, 1979.
3. H. Bordihn, J. Dassow, and M. Holzer. Extending regular expressions with homomorphic replacement. *RAIRO Theoretical Informatics and Applications*, 44:229–255, 2010.
4. C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *Int. Journal of Foundations of Computer Science*, 14:1007–1018, 2003.
5. C. Câmpeanu and N. Santean. On the intersection of regex languages with regular languages. *Theoretical Computer Science*, 410:2336–2344, 2009.
6. C. Câmpeanu and S. Yu. Pattern expressions and pattern automata. *Information Processing Letters*, 92:267–274, 2004.
7. B. Carle and P. Narendran. On extended regular expressions. In *Proc. LATA 2009*, volume 5457 of *LNCS*, pages 279–289, 2009.
8. J. Dassow and G. Păun. *Regulated Rewriting in Formal Language Theory.* Springer-Verlag Berlin, 1989.
9. J. Dassow, G. Păun, and A. Salomaa. Grammars with controlled derivations. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2, pages 101–154. Springer, 1997.
10. R. W. Floyd. On the nonexistence of a phrase structure grammar for algol 60. *Communications of the ACM*, 5:483–484, 1962.
11. D. D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory of Computing Systems*, 53:159–193, 2013.
12. J. E. F. Friedl. *Mastering Regular Expressions.* O'Reilly, Sebastopol, CA, third edition, 2006.
13. L. Kallmeyer. *Parsing Beyond Context-Free Grammars.* Springer-Verlag, 2010.
14. L. Kari, G. Rozenberg, and A. Salomaa. L systems. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 5, pages 253–328. Springer, 1997.
15. G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Synchronized regular expressions. *Acta Informatica*, 39:31–70, 2003.
16. M. L. Schmid. Inside the class of regex languages. *International Journal of Foundations of Computer Science*, 24:1117–1134, 2013.
17. S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, chapter 2, pages 41–110. Springer, 1997.