Δ-Clearing Restarting Automata and CFL[*] Technical Report

Peter Černo František Mráz

April 28, 2011

Abstract

Δ-clearing restarting automata represent a new restricted model of restarting automata which, based on a limited context, can either delete a substring of the current content of its tape or replace a substring by a special auxiliary symbol Δ, which cannot be overwritten anymore, but it can be deleted later. The main result of this paper consists in proving that besides their limited operations, Δ-clearing restarting automata recognize all context-free languages.

Keywords: analysis by reduction, context-free languages, Δ-clearing restarting automata, formal languages.

1 Introduction

Restarting automata [5] were introduced as a tool for modeling some techniques used for natural language processing. In particular they are used for analysis by reduction which is method for checking (syntactical) correctness or non-correctness of a sentence. While restarting automata are quite general (see [10] for an overview), they still lack some properties which could facilitate their wider use. One of such properties is that they differ from the classical models of automata used in parsing. Restarting automata work in cycles. They iteratively simplify the input sentence while preserving its (non-)correctness until a simple (short) sentence is obtained for which it is easy to decide its (non-)correctness. If the obtained simple form is correct, the whole input is accepted too. Each simplification is done within a so-called cycle. During one cycle, a restarting automaton scans the current input from the left to the right using a fixed size scanning window and a finite state control. In one of the steps of a cycle the automaton rewrites the content of its scanning window by a shorter string and later restarts its computation.

[*]This work was partially supported by the Grant Agency of Charles University under Grant-No. 272111/A-INF/MFF and by the Czech Science Foundation under Grant-No. P103/10/0783 and Grant-No. P202/10/1333.

Recently, Kutrib et al. in [6] and [7] introduced stateless restarting automata. For monotone and/or deterministic version of these automata, if they can use auxiliary symbols in rewriting, then they have the same power as the corresponding versions with states ([6, 8]). However, the stateless versions of restarting automata without auxiliary symbols are strictly weaker than the respective versions which can use states.

Černo and Mráz [2] introduced an even more simplified version of restarting automata called clearing restarting automata. While general restarting automata see the whole part of the current sentence (word) to the left (and possibly also to the right) of the place they rewrite, the rewriting done by clearing restarting automata depends only on a fixed context around the rewritten subword. Moreover, clearing restarting automata can only "clear" a subword, i.e. completely delete a subword based on limited context around the "cleared" subword. Hence the automata have no states and in one cycle they can rewrite (exactly once) at any place according to some of their finitely many instructions. Obviously, such automata are more restricted than the weakest version of the stateless restarting automata (the so-called stateless R-automata). It turned out that clearing restarting automata are rather limited. While they can recognize all regular languages and even some languages that are not context-free, they cannot recognize all context-free languages (see [2]). Hence there were introduced Δ-clearing automata and Δ^*-clearing automata that can use an auxiliary symbol Δ. Besides deleting a subword, Δ-clearing automata can rewrite a subword by the special symbol Δ, which can be deleted in later cycles, too. Δ^*-clearing automata are even stronger, as they can also rewrite a subword by Δ^i, where i is not greater than the length of the rewritten word.

In [1] we have shown that Δ-clearing automaton can accept the Greibach's "hardest context-free language" and later in [2] there was shown that Δ^*-clearing automata can recognize all context-free languages. [2] conjectured that also Δ-clearing automata can recognize CFL. In this paper we prove the conjecture.

The paper is divided into several sections. In Section 2 we introduce a general concept called *context rewriting system* which will serve us as a framework for Δ-clearing restarting automata and their extended version Δ^*-clearing restarting automata. We also prove some of their basic properties. The main source for this section is [2]. In Section 3 we provide a different algorithmic viewpoint on clearing restarting automata which will later simplify many technical constructions used in this report. In Section 4 we describe a special coding used by Δ-clearing restarting automata to encode some information into its tape. In Sections 5 and 6 we describe the algorithm behind the Δ-clearing restarting automaton recognizing a given context-free language.

We use the standard notation from the theory of automata and formal languages. An *alphabet* is a finite nonempty set. The elements of an alphabet Σ are called *letters* or *symbols*. A *word* or *string* over an alphabet Σ is a finite sequence consisting of zero or more letters of Σ, whereby the same letter may occur several times. The sequence of zero letters is called the *empty word*, written λ. The set of all words (all nonempty words, respectively) over an alphabet Σ is denoted by Σ^* (Σ^+, respectively). If x and y are words over Σ, then so is their *catenation* (or *concatenation*) xy (or $x \cdot y$), obtained by juxtaposition, that is, writing x and y one after another. Catenation is an associative operation and the empty word λ acts as an identity: $w\lambda = \lambda w = w$ holds for all words w. Because of the associativity, we may use the notation w^i in the usual way. By definition, $w^0 = \lambda$.

Let u be a word in Σ^*, say $u = a_1 \dots a_n$ with $a_i \in \Sigma$. We say that n is the *length* of u and we write $|u| = n$. The sets of all words over Σ of length k, or at most k, are denoted by Σ^k and $\Sigma^{\le k}$, respectively. Finally a *factorization* of u is any sequence u_1, ..., u_t of words such that $u = u_1 \cdots u_t$.

For a pair u, v of words we define the following relations:

1. u is a *prefix* of v, if there exists a word z such that $v = uz$,
2. u is a *suffix* of v, if there exists a word z such that $v = zu$, and
3. u is a *factor* (or *subword*) of v, if there exist words z and z' such that $v = zuz'$.

Observe that u itself and λ are subwords, prefixes and suffixes of u. Other subwords, prefixes and suffixes are called *proper*.

Subsets, finite or infinite, of Σ^* are referred to as (*formal*) *languages* over Σ.

2 Theoretical Background

In this section we introduce a general concept called *context rewriting system* which will serve us as a framework for Δ-clearing restarting automata and their extended version Δ^*-clearing restarting automata.

Definition 2.1 ([2]). *Let k be a positive integer. A k-*context rewriting system *(k-*CRS *for short) is a system $R = (\Sigma, \Gamma, I)$, where Σ is an input alphabet, $\Gamma \supseteq \Sigma$ is a working alphabet not containing the special symbols ¢ and \$, called* sentinels*, and I is a finite set of* instructions *of the form:*

$$(x, z \to t, y) \ ,$$

where x is called left context*, $x \in LC_k = \Gamma^k \cup ¢ \cdot \Gamma^{\le k-1}$, y is called* right context*, $y \in RC_k = \Gamma^k \cup \Gamma^{\le k-1} \cdot \$$ and $z \to t$ is called* rule*, $z, t \in \Gamma^*$.*

A word $w = uzv$ can be rewritten *into utv (denoted as $uzv \to_R utv$) if and only if there exists an instruction $i = (x, z \to t, y) \in I$ such that x is a suffix of $¢ \cdot u$ and y is a prefix of $v \cdot \$$. We often underline the rewritten part of the word w, and if the instruction i is known we use $\to_R^{(i)}$ instead of $\to_R$, i.e. $u\underline{z}v \to_R^{(i)} utv$. The relation $\to_R \subseteq \Gamma^* \times \Gamma^*$ is called* rewriting relation.

The production language *(*reduction language*, respectively) associated with R is defined as $L^+(R) = \{w \in \Sigma^* \mid \lambda \to_R^* w\}$ ($L^-(R) = \{w \in \Sigma^* \mid w \to_R^* \lambda\}$, respectively), where $\to_R^*$ is the reflexive and transitive closure of $\to_R$. Note that, by definition, $\lambda \in L^+(R)$ ($\lambda \in L^-(R)$, respectively).*

The production characteristic language *(*reduction characteristic language*, respectively) associated with R is defined as $L_C^+(R) = \{w \in \Gamma^* \mid \lambda \to_R^* w\}$ ($L_C^-(R) = \{w \in \Gamma^* \mid w \to_R^* \lambda\}$, respectively). Similarly, by definition, $\lambda \in L_C^+(R)$ ($\lambda \in L_C^-(R)$, respectively). Obviously, for each k-*CRS *R, it holds $L^+(R) = L_C^+(R) \cap \Sigma^*$ ($L^-(R) = L_C^-(R) \cap \Sigma^*$).*

Remark 2.1. *We extend Definition 2.1 with the following notation: if $X \subseteq LC_k$ and $Y \subseteq RC_k$ are finite nonempty sets, and Z is a finite nonempty set of rules of the form $z \to t$, $z, t \in \Gamma^*$, then we define $(X, Z, Y) = \{(x, z \to t, y) \mid x \in X, (z \to t) \in Z, y \in Y\}$. However, if $X = \{x\}$, then instead of writing $(\{x\}, Z, Y)$ we write only (x, Z, Y) for short. The same holds for the sets Z and Y, too.*

By reversing all rewriting rules of a k-CRS we obtain a *dual system.*

Definition 2.2 ([2])**.** *Let $R = (\Sigma, \Gamma, I)$ be a k-CRS. A* dual context rewriting system *R^D is a k-CRS $R^D = (\Sigma, \Gamma, I^D)$, where $I^D = \{(x, t \to z, y) \mid (x, z \to t, y) \in I\}$. For an instruction $i = (x, z \to t, y)$, we call $i^D = (x, t \to z, y)$ a* dual instruction *to the instruction i. We also define a* dual rewriting relation *to the relation $\to_R$ as $(\to_R)^D = \to_{R^D}$.*

Theorem 2.1 (Duality theorem [2])**.** *For each k-CRS $R = (\Sigma, \Gamma, I)$ and its corresponding dual system R^D the following holds:*

$$\begin{array}{ll} (1) & (\to_R)^D = (\to_R)^{-1}, \\ (2) & (R^D)^D = R, \\ (3) & L^+(R) = L^-(R^D), \\ (4) & L_C^+(R) = L_C^-(R^D). \end{array}$$

Proof. (1) Obviously, for all $w, w' \in \Gamma^* : w(\to_R)^D w' \Leftrightarrow w \to_{R^D} w' \Leftrightarrow w' \to_R w$, thus $(\to_R)^D = (\to_R)^{-1}$.
(2) is trivial, (3) and (4) follow from (1). □

Naturally, if we increase the length of contexts used in instructions of a CRS, we do not decrease their expressiveness.

Theorem 2.2 (Context extension theorem ([2]))**.** *For each k-CRS $R = (\Sigma, \Gamma, I)$ there exists a $(k+1)$-CRS $R' = (\Sigma, \Gamma, I')$ such that, for each $w, w' \in \Gamma^*$, it holds $w \to_R w' \Leftrightarrow w \to_{R'} w'$. Moreover, both R and R' use the same rewriting rules:*

$$\{z \to t \mid (x, z \to t, y) \in I\} = \{z' \to t' \mid (x', z' \to t', y') \in I'\} .$$

Proof. For each instruction $i = (x, z \to t, y) \in I$ let us define J_i to be $(X, z \to t, Y)$, where:
(1) If $x \in \Gamma^k$, then $X = (\Gamma \cup \{¢\}) \cdot x$. If $x \in ¢ \cdot \Gamma^{\le k-1}$, then $X = \{x\}$. Evidently, $X \subseteq LC_{k+1}$.
(2) If $y \in \Gamma^k$, then $Y = y \cdot (\Gamma \cup \{\$\})$. If $y \in \Gamma^{\le k-1} \cdot \$$, then $Y = \{y\}$. Obviously, $Y \subseteq RC_{k+1}$.
It is easy to see that $u\underline{z}v \to^{(i)} utv$ if and only if $u\underline{z}v \to^{(j)} utv$ for some $j \in J_i$. This implies that if we set $I' := \bigcup_{i \in I} J_i$, then we get a $(k+1)$-CRS $R' = (\Sigma, \Gamma, I')$ which has the same rewriting relation as the k-CRS $R = (\Sigma, \Gamma, I)$ and both R and R' use the same rewriting rules. □

Remark 2.2. *Based on the above result, in Definition 2.1 we can allow contexts of any length up to k, i.e. we can use:*
$LC_{\le k} = \Gamma^{\le k} \cup ¢ \cdot \Gamma^{\le k-1} = \bigcup_{i \le k} LC_i$ instead of LC_k and
$RC_{\le k} = \Gamma^{\le k} \cup \Gamma^{\le k-1} \cdot \$ = \bigcup_{i \le k} RC_i$ instead of RC_k.

The following theorem corresponds to correctness and error preserving properties of restating automata.

Theorem 2.3 (Correctness and error preserving theorem [2])**.** *Let $R = (\Sigma, \Gamma, I)$ be a k-*CRS *and u, v be two words from Σ^* such that $u \to_R^* v$. Then:*

$$\begin{array}{ll} (1) & u \in L^+(R) \Rightarrow v \in L^+(R), \\ (2) & u \notin L^-(R) \Rightarrow v \notin L^-(R). \end{array}$$

Proof. Let us suppose that $u, v \in \Sigma^*$ and $u \to_R^* v$.
(1) $u \in L^+(R)$ implies $\lambda \to_R^* u$ and thus $\lambda \to_R^* u \to_R^* v$. Hence, v is in $L^+(R)$.
(2) $v \in L^-(R)$ implies $v \to_R^* \lambda$ and thus $u \to_R^* v \to_R^* \lambda$, which implies $u \in L^-(R)$. □

It is easy to see that general k-CRS can simulate any type 0 grammar (according to the Chomsky hierarchy [4]). Hence we will not study k-CRS in their general form, since they are too powerful (they can represent recursively enumerable languages). Instead, we will always put some restrictions on the rules of instructions and then study such restricted models. The first model we introduce is called *clearing restarting automaton* which is a k-CRS such that $\Sigma = \Gamma$ and all rules in its instructions are of the form $z \to \lambda$, where $z \in \Sigma^+$.

Definition 2.3 ([2])**.** *Let k be a positive integer. A k-*clearing restarting automaton *(k-*cl-RA *for short) is a system $M = (\Sigma, I)$, where $R = (\Sigma, \Sigma, I)$ is a k-*CRS *such that for each instruction $i = (x, z \to t, y) \in I$ it holds $z \in \Sigma^+$ and $t = \lambda$. Since t is always the empty word, we use the notation $i = (x, z, y)$. The* width *of the instruction $i = (x, z, y)$ is $|i| = |xzy|$.*

*The k-*cl-RA *M* recognizes *the language $L(M) = \{w \in \Sigma^* \mid w \vdash_M^* \lambda\} = L^-(M)$, where $\vdash_M$ is the* rewriting relation *$\to_R$ of R.*

In the following, k-cl-RA (cl-RA, respectively) denotes the class of all k-clearing restarting automata (clearing restarting automata, respectively), where cl-RA $= \bigcup_{k=1}^{\infty} k$-cl-RA. $\mathcal{L}(k$-cl-RA$)$ ($\mathcal{L}($cl-RA$)$, respectively) denotes the class of all languages accepted by k-cl-RA (cl-RA, respectively), $\mathcal{L}($cl-RA$) = \bigcup_{k=1}^{\infty} \mathcal{L}(k$-cl-RA$)$.

The simplicity of the cl-RA model implies that the investigation of its properties and the proofs are not so difficult and also the learning of languages is easy, fast and straightforward. Another important advantage of this model is that the instructions are human readable and simpler than the meta-instructions of general restarting automata [10].

Example 2.1. *Let $M = (\Sigma, I)$ be the* 1-cl-RA *with $\Sigma = \{a, b\}$ and I consisting of the following two instructions:*

$$\begin{array}{ll} (1) & (a, ab, b), \\ (2) & (¢, ab, \$). \end{array}$$

Then we have $aaa\underline{ab}bbb \vdash_M^{(1)} aa\underline{ab}bb \vdash_M^{(1)} a\underline{ab}b \vdash_M^{(1)} \underline{ab} \vdash_M^{(2)} \lambda$ which means that $aaaabbbb \vdash_M^* \lambda$. So the word $aaaabbbb$ is accepted by M. It is easy to see that M recognizes the language $L(M) = \{a^n b^n \mid n \geq 0\}$.

Remark 2.3. *By definition, each* cl-RA *accepts λ. If we say that a* cl-RA *M* recognizes *(or* accepts*) a language L, we always mean that $L(M) = L \cup \{\lambda\}$.*

This implicit acceptance of the empty word can be avoided by a slight modification of the definition of clearing restarting automata, or even context rewriting systems, but in principle, we would not get a more powerful model.

Remark 2.4. *As we have seen, the language recognized by a k-*cl-RA *$M = (\Sigma, I)$ is defined as the reduction language of M, i.e. $L(M) = L^{-}(M)$. Also note that $L^{+}(M) = \{\lambda\}$. Now suppose that $N = M^{D}$ is a dual k-*CRS *to the k-*CRS *M. N is no longer a clearing restarting automaton, because it contains instructions of the form $(x, \lambda \to z, y)$, where $z \in \Sigma^{+}$. But according to the Duality Theorem (Theorem 2.1), $L(M) = L^{-}(M) = L^{+}(M^{D}) = L^{+}(N)$. This reasoning suggests that we can look at clearing restarting automata from two points of view:*

1. *We can consider a* cl-RA *$M = (\Sigma, I)$ to be an automaton that recognizes the language $L(M)$ by using* reductions*, i.e. $L(M) = \{w \in \Sigma^{*} \mid w \vdash_{M}^{*} \lambda\}$, where $\vdash_{M}$ is the rewriting relation of M, called* reduction relation*.*
2. *We can consider a* cl-RA *$M = (\Sigma, I)$ to be a generative device generating the language $L(M)$ by using* productions*, i.e. $L(M) = \{w \in \Sigma^{*} \mid \lambda \dashv_{M}^{*} w\}$, where $\dashv_{M} = (\vdash_{M})^{-1}$ is the rewriting relation of $N = M^{D}$, called* production relation*.*

Clearing restarting automata are studied in [2]. We only mention that they can recognize all regular languages, some context-free languages and even some non-context-free languages. However, there are some context-free languages that are outside the class $\mathcal{L}$(cl-RA).

Theorem 2.4. *The language $L = \{a^{n}cb^{n} \mid n \geq 0\} \cup \{\lambda\}$ is not recognized by any* cl-RA*.*

Proof. For a contradiction, let us suppose that there exists a k-cl-RA $M = (\Sigma, I)$ such that $L(M) = L$. Let m be the maximal width of instructions of M. Obviously, $a^{m}cb^{m} \in L$ implies $a^{m}cb^{m} \vdash_{M}^{*} \lambda$ and the word $a^{m}cb^{m}$ cannot be reduced to λ in a single step. On the other hand, if we erase any single nonempty continuous proper subword from the word $a^{m}cb^{m}$, then we get a word that does not belong to L – a contradiction to $L(M) = L$. $\square$

In [2] there were introduced two extended versions of clearing restarting automata – the so-called Δ-clearing restarting automata and Δ^{*}-clearing restarting automata. Both of them can use a single auxiliary symbol Δ only. Δ-clearing restarting automata can leave a mark – a symbol Δ – at the place of deleting besides rewriting into the empty word λ. Δ^{*}-clearing restarting automata can rewrite a subword w into Δ^{k} where k is bounded from above by the length of w. In what follows we will first repeat a result from [2] that shows Δ^{*}-clearing restarting automata are powerful enough to recognize all context-free languages. Actually, the proof we will present differs in some parts from the proof from [2]. In particular, we will use a modified encoding of nonterminals, which will be later used in the next part of the paper where we describe how to transform a Δ^{*}-clearing restarting automaton recognizing a given context-free language into a Δ-clearing restarting automaton recognizing the same language.

Definition 2.4. *Let k be a positive integer. A* k-Δ-clearing restarting automaton *(k-Δcl-RA for short) is a system $M = (\Sigma, I)$, where $R = (\Sigma, \Gamma, I)$ is a k-CRS such that $\Delta \notin \Sigma$, $\Gamma = \Sigma \cup \{\Delta\}$, and for each instruction $i = (x, z \to t, y) \in I$: $z \in \Gamma^+$ and either $t = \lambda$, or $t = \Delta$.*

The k-Δcl-RA M recognizes *the language $L(M) = \{w \in \Sigma^* \mid w \vdash_M^* \lambda\} = L^-(M)$, where $\vdash_M$ is the* rewriting relation *$\to_R$ of R.*

The characteristic language *of M is the language $L_C(M) = L_C^-(M)$.*

By Δcl-RA we denote the class of all Δ-clearing restarting automata. $\mathcal{L}(k\text{-}\Delta\text{cl-RA})$ ($\mathcal{L}(\Delta\text{cl-RA})$, respectively) denotes the class of all languages accepted by k-Δcl-RA (Δcl-RA, respectively).

Example 2.2. *Let $M = (\Sigma, I)$ be a 1-Δcl-RA with $\Sigma = \{a, b, c\}$ and the set of instructions I consisting of the following instructions:*

$$\begin{array}{ll} (1) & (a, c \to \Delta, b), \\ (2) & (a, a\Delta b \to \Delta, b), \\ (3) & (¢, a\Delta b \to \Delta, \$), \\ (4) & (¢, c \to \Delta, \$), \\ (5) & (¢, \Delta \to \lambda, \$). \end{array}$$

An input word a^ncb^n, for arbitrary $n > 1$, is accepted by M in the following way:

$$a^n\underline{c}b^n \vdash_M^{(1)} a^{n-1}\underline{a\Delta b}b^{n-1}n \vdash_M^{(2)} a^{n-1}\Delta b^{n-1} \vdash_M^{(2)} \ldots \vdash_M^{(2)} \underline{a\Delta b} \vdash_M^{(3)} \underline{\Delta} \vdash_M^{(5)} \lambda\ .$$

First, M deletes c while marking its position by Δ. In each of the following steps, M deletes one a and one b around Δ until it obtains single-letter word Δ, which is then reduced into the empty word λ.

It is easy to see that M recognizes the language $L = \{a^ncb^n \mid n \geq 0\} \cup \{\lambda\}$.

The characteristic language of M is

$$L_C(M) = \{a^ncb^n, a^n\Delta b^n \mid n \geq 0\} \cup \{\lambda\}\ .$$

Now we introduce a generalization of Δcl-RA, a so-called Δ^**-clearing restarting automata*, which are able to recognize all context-free languages.

Definition 2.5. *Let k be a positive integer. A* k-Δ^*-clearing restarting automaton *(k-Δ^*cl-RA for short) is a system $M = (\Sigma, I)$, where $R = (\Sigma, \Gamma, I)$ is a k-CRS such that $\Delta \notin \Sigma$, $\Gamma = \Sigma \cup \{\Delta\}$, and for each instruction $i = (x, z \to t, y) \in I$: $z \in \Gamma^+$ and $t = \Delta^i$, where $0 \leq i \leq |z|$.*

The k-Δ^*cl-RA M recognizes *the language $L(M) = \{w \in \Sigma^* \mid w \vdash_M^* \lambda\} = L^-(M)$, where $\vdash_M$ is the* rewriting relation *$\to_R$ of R.*

The characteristic language *of M is the language $L_C(M) = L_C^-(M)$.*

By Δ^*cl-RA we denote the class of all Δ^*-clearing restarting automata. $\mathcal{L}(k\text{-}\Delta^*\mathsf{cl\text{-}RA})$ ($\mathcal{L}(\Delta^*\mathsf{cl\text{-}RA})$, respectively) denotes the class of all languages accepted by k-Δ^*cl-RA (Δ^*cl-RA, respectively).

Next we show that 1-Δ^*cl-RA can recognize any context-free language. The following proof utilizes the same idea as in [2], but a little modified encoding of nonterminals of a context-free grammar using Δ's.

Theorem 2.5. *For each context-free language L there exists a* 1-Δ^*cl-RA-*automaton M recognizing L.*

Proof. Let L be a context-free language. Then there exists a context-free grammar $G = (V_N, V_T, S, P)$ in Chomsky normal form generating the language $L(G) = L \smallsetminus \{\lambda\}$. Let $V_N = \{N_1, \ldots, N_m\}$, $S = N_1$ and $\Delta \notin V_N \cup V_T$, and let $G' = (V_N, V_T', S, P')$ be the grammar obtained from G by adding a new terminal symbol Δ to V_T ($V_T' = \Sigma \cup \{\Delta\}$), and adding new productions $N_i \to a\Delta^i b$ to P, for all $1 \le i \le m$ and all $a, b \in V_T$. Obviously, $L(G') \cap \Sigma^* = L(G)$. We will show that we can effectively construct a 1-Δ^*cl-RA M such that $L_C(M) = L(G') \cup \{\lambda\}$ and $L(M) = L_C(M) \cap \Sigma^* = (L(G') \cup \{\lambda\}) \cap \Sigma^* = L(G) \cup \{\lambda\}$.

For the automaton M all the words $a\Delta^i b$ for all $a, b \in \Sigma$ represent "codes" for the nonterminal N_i. The letters $a, b \in \Sigma$ serve as separators for distinguishing several consecutive encoded nonterminals.

The automaton M works in a bottom-up manner. If the automaton recognizes that some subword w of the input tape can be derived from some nonterminal N_i, then the automaton can (nondeterministically) replace this subword w by a corresponding code Δ^i. Or to be more precise, the automaton M replaces only the inner part of the subword w by the code Δ^i in order to leave the first and the last letter of w as separators. If the word on the input tape is short enough and belongs to the language $L(G')$ then the automaton M just erases the whole input word in a single step.

The obvious obstacle of this approach is how to ensure that the resulting automaton M will have only finitely many instructions? The answer lies in the observation that for every context-free grammar there exists an upper limit c for the length of subwords w such that if we restrict the automaton M only to the words of length at most c, then the automaton will work correctly and recognize exactly the corresponding context-free language. Before we continue we prove the following useful lemma.

Lemma 2.1 (Tree Lemma). *Let T be a rooted binary tree with a root node r, such that each leaf node l of T has an associated weight $w(l) \in \{1, 2, \ldots, U\}$ (where U is a positive integer constant) and each internal node v of T has weight $w(v)$ equal to the sum of weights of all its descendant leaf nodes. Then for any positive integer constant $c \ge U$ either $w(r) \le c$, or there is an internal node v such that $c < w(v) \le 2c$.*

Proof. If $w(r) \le c$ then there is nothing to prove and the lemma obviously holds. If $w(r) > c$ then we inductively define a path $v_1 v_2 \ldots v_k$ in the tree T, such that $k > 1$, $v_1, v_2, \ldots, v_{k-1}$ are internal nodes and v_k is a leaf node. First, we define $v_1 = r$. The node v_1 cannot be a leaf node, since $w(v_1) = w(r) > c \ge U$. Let us suppose that we have defined the nodes

$v_1, v_2, \ldots, v_i$. If v_i is a leaf node, then $k = i$ and the path is completed. Otherwise, v_i is not a leaf node and it has two sons (the tree is binary) v_l and v_r. Then we define v_{i+1} to be a son with weight at least half of the weight of v_i. More precisely, if $w(v_\mathrm{l}) > w(v_\mathrm{r})$, then $v_{i+1} = v_\mathrm{l}$, otherwise $v_{i+1} = v_\mathrm{r}$.

Observe, that $w(v_1) \geq w(v_2) \geq \ldots \geq w(v_k)$ and $w(v_k) \leq U \leq c$. Let $j \in \{1, 2, \ldots, k-1\}$ be the largest index such that $w(v_j) > c$. We claim that $w(v_j) \leq 2c$. Obviously, $j < k$ and v_j has two sons v_l and v_r. Without loss of generality we can suppose that $w_{j+1} = v_\mathrm{l}$, i.e. $w(v_\mathrm{l}) \geq w(v_\mathrm{r})$. By the choice of j we have $w(v_{j+1}) = w(v_\mathrm{l}) \leq c$, i.e. $w(v_j) = w(v_\mathrm{l}) + w(v_\mathrm{r}) \leq 2w(v_\mathrm{l}) \leq 2c$. □

Now we apply Tree Lemma 2.1 to our context-free grammar G'. Consider any word $w \in L(G')$ and any derivation tree T corresponding to this word w. The nonterminals in the derivation tree T represent internal nodes and the terminal words derived from nonterminals in the derivation tree T represent leaf nodes. Let r be the root of T labeled by the initial nonterminal S. If we define the weight of the leaf nodes as the length of the words represented by these leaf nodes we obtain a binary tree with an upper limit $U = m + 2$ for the weight of leaf nodes, where m is the number of nonterminals in G'. Let us take $c = U = m + 2$. By Tree Lemma 2.1 either $w(r) \leq c$, or there is an internal node v such that $c < w(v) \leq 2c$. In other words, either $|w| \leq c$, or there exists a derivation $S \Rightarrow^*_{G'} xN_iy \Rightarrow^*_{G'} xzy$ such that $w = xzy$ and $c < |z| \leq 2c$. Thus we have shown the following corollary.

Corollary 2.1. *Let $G' = (V_N, V'_T, S, P')$ be the grammar constructed above and w be a word from $L(G')$ of length $|w| > c = |V_N| + 2$. Then there exist words x, y, z from $(V'_T)^*$ and a nonterminal $N_i \in V_N$ such that $S \Rightarrow^*_{G'} xN_iy \Rightarrow^*_{G'} xzy$ and $w = xzy$.*

Now we construct a 1-Δ^*cl-RA $M = (\Sigma, I)$, where $\Sigma = V_T$, $\Gamma = \Sigma \cup \{\Delta\} = V'_T$, such that $L_C(M) = L(G') \cup \{\lambda\}$. First, we set:

$$I_1 = \{(¢, w \to \lambda, \$) \mid w \in L(G'), |w| \leq c\}.$$

For every $i \in \{1, 2, \ldots, m\}$ let us define:

$$L_i = \{z \in \Gamma^* \mid N_i \Rightarrow^*_{G'} z, c < |z| \leq 2c\}.$$

For every such $z \in L_i$, $z = z_1 z_2 \ldots z_{s-1} z_s$, consider the instruction:

$$(z_1, z_2 \ldots z_{s-1} \to \Delta^i, z_s)\,.$$

This instruction rewrites the inner part of the word z to Δ^i leaving z_1 and z_s as separators. Let I_2 be the set of all such instructions. (Observe that $z_1, z_s \in \Sigma$, and both I_1 and I_2 are finite sets of instructions). Then $M = (\Sigma, I_1 \cup I_2)$ is the required automaton.

Lemma 2.2. *$L(G') \subseteq L_C(M)$.*

Proof. (By induction on the length of words from $L(G')$)

Let $w \in L(G')$. If $|w| \leq c$, then $(¢, w \to \lambda, \$) \in I_1$, thus $w \vdash_M \lambda$ which implies $w \in L_C(M)$. Suppose $|w| > c$. According to Corollary 2.1, there are $x, z, y \in \Gamma^*$, $c < |z| \leq 2c$, and $i \in \{1, 2, \ldots, m\}$, such that there exists a derivation $S \Rightarrow^*_{G'} xN_iy \Rightarrow^*_{G'} xzy = w$, where $z = z_1z_2 \ldots z_{s-1}z_s$ for some $z_1, \ldots, z_s \in V'_T$. The definition of I_2 implies that there is an instruction $(z_1, z_2 \ldots z_{s-1} \to \Delta^i, z_s) \in I_2$. If we use this instruction in the word w, we get the reduction: $w = xz_1\underline{z_2 \ldots z_{s-1}}z_sy \vdash_M xz_1\Delta^iz_sy = w'$. Now $s = |z| > c = m + 2$ implies that $s - 2 > m \geq i$. So $|z_2 \ldots z_{s-1}| > |\Delta^i|$ implies $|w| > |w'|$. On the other hand, in G' there is a production rule $N_i \to z_1\Delta^iz_s$, so $S \Rightarrow^*_{G'} xN_iy \Rightarrow_{G'} xz_1\Delta^iz_sy = w'$. Thus $w' \in L(G')$ and by the induction hypothesis (since $|w'| < |w|$) we have $w' \in L_C(M)$ and $w' \vdash^*_M \lambda$ which implies $w \vdash_M w' \vdash^*_M \lambda$ and $w \in L_C(M)$. $\square$

Lemma 2.3. $L_C(M) \subseteq L(G') \cup \{\lambda\}$.

Proof. (By induction on the number of reduction steps)

Suppose $w \in L_C(M)$ and $w = w_n \vdash_M w_{n-1} \vdash_M \ldots \vdash_M w_1 \vdash_M \lambda$. For each $i = 1, 2, \ldots n$, let us prove that $w_i \in L(G') \cup \{\lambda\}$. $w_1 \vdash_M \lambda$ implies that there is the instruction $(¢, w_1 \to \lambda, \$)$ in I_1, and thus $w_1 \in L(G')$ according to the definition of I_1. Suppose that $w_j \in L(G')$ and in the reduction $w_{j+1} \vdash_M w_j$ we have used the instruction $\phi = (z_1, z_2 \ldots z_{s-1} \to \Delta^i, z_s) \in I_2$, i.e. $w_{j+1} = xz_1\underline{z_2 \ldots z_{s-1}}z_sy \vdash_M xz_1\Delta^iz_sy = w_j$. We have $w_j \in L(G')$, therefore $S \Rightarrow^*_{G'} xz_1\Delta^iz_sy$. The sequence of letters $z_1\Delta^iz_s$ in our derivation could have been created only by using the production rule $N_i \to z_1\Delta^iz_s$ (because $z_1, z_s \in \Sigma$). Therefore, there exists a derivation $S \Rightarrow^*_{G'} xN_iy \Rightarrow_{G'} xz_1\Delta^iz_sy$ in G'. From the definition of $\phi \in I_2$ we also have $N_i \Rightarrow^*_{G'} z_1z_2 \ldots z_{s-1}z_s$, where $c < s = |z| \leq 2c$. Thus in G' there exists a derivation $S \Rightarrow^*_{G'} xN_iy \Rightarrow^*_{G'} xz_1z_2 \ldots z_{s-1}z_sy = w_{j+1}$, which implies that $w_{j+1} \in L(G')$. $\square$

This completes the proof of Theorem 2.5. $\square$

We have shown that 1-Δ^*cl-RA are able to recognize all context-free languages (containing the empty word λ – see Remark 2.3). This result opens an interesting question whether it is possible to transform each Δ^*cl-RA into an equivalent Δcl-RA. If we are interested only in the problem whether Δcl-RA can recognize all context-free languages, then we do not need to do this transformation to all Δ^*cl-RA. We just need to do this transformation to such Δ^*cl-RA which were obtained from a context-free grammar, as was shown above. Moreover, the aforementioned construction can be generalized, i.e. we can put some extra restrictions on the instructions of the resulting Δ^*cl-RA. We can generalize the construction of the grammar G' and the corresponding Δ^*cl-RA M in the following four ways:

1. We can choose a minimal length $m_0 \geq 1$ of codes for nonterminals, i.e. we code N_i by using at least m_0 consecutive letters Δ.

2. We can choose a minimal length $m_1 \geq 1$ of shortening for each reduction, i.e. for each instruction $(x, u \to \Delta^r, y)$ such that $r \geq 1$, we guarantee that $|u| - |\Delta^r| \geq m_1$.

3. We can choose a number of codes $m_2 \geq 1$ representing one nonterminal, i.e. we code N_i by using $m_0 + m_2(i-1) + j - 1$ consecutive letters Δ, for all $j \in \{1, 2, \dots m_2\}$.
4. We can choose a length $k \geq 1$ of the separator, i.e. instead of one letter we use k consecutive arbitrary letters from Σ as a separator.

Suppose we have chosen the constants m_0, m_1, m_2 and k, and L be a given context-free language. Let $G = (V_N, V_T, S, P)$ be a context-free grammar in Chomsky normal form generating the language $L(G) = L \setminus \{\lambda\}$, $V_N = \{N_1, \dots, N_m\}$, $S = N_1$ and $\Delta \notin V_N \cup V_T$. Let $G' = (V_N, V_T', S, P')$ be a grammar which we obtain from G by adding a new terminal symbol Δ to V_T and adding new productions $N_i \to x\Delta^{m_0+m_2(i-1)+j-1}y$ to P, for all $1 \leq i \leq m$, $1 \leq j \leq m_2$, and all $x, y \in (V_T)^k$. Let $\Sigma = V_T$ and $\Gamma = \Sigma \cup \{\Delta\} = V_T'$. We can effectively construct a k-Δ^*cl-RA M with the above properties such that $L_C(M) = L(G') \cup \{\lambda\}$. This again implies that $L(M) = L_C(M) \cap \Sigma^* = L(G) \cup \{\lambda\}$.

Consider any word $w \in L(G')$ and any derivation tree T corresponding to this word w. Again, we can look at T as a binary tree with internal nodes corresponding to nonterminals in T, and leaf nodes corresponding to terminal words in T. Let r be the root internal node corresponding to the root nonterminal S in the derivation tree T. If we define the weight of leaf nodes as the length of the words represented by these leaf nodes we obtain a binary tree with an upper limit $U = m_0 + m_2m - 1 + 2k$ for the weight of the leaf nodes (because the largest possible code is the code $x\Delta^{m_0+m_2(m-1)+m_2-1}y$ for the nonterminal N_m, where $|x| = |y| = k$). Let us take $c = U + m_1 - 1$. By Tree Lemma 2.1 either $w(r) \leq c$, or there is an internal node v such that $c < w(v) \leq 2c$. In other words, either $|w| \leq c$, or there exists a derivation $S \Rightarrow_{G'}^* xN_iy \Rightarrow_{G'}^* xzy$ such that $w = xzy$ and $c < |z| \leq 2c$. Now we can construct a k-Δ^*cl-RA M as follows. First, we set:

$$I_1 = \{(¢, w \to \lambda, \$) \mid w \in L(G'), |w| \leq c\}.$$

For every $i \in \{1, 2, \dots, m\}$ let us define:

$$L_i = \{z \in \Gamma^* \mid N_i \Rightarrow_{G'}^* z, c < |z| \leq 2c\}.$$

For every such $z \in L_i$, $z = xuy$, $|x| = |y| = k$, and every $j \in \{1, 2, \dots, m_2\}$ consider the following instruction ϕ (see Figure 1):

$$\phi = (x, u \to \Delta^{m_0+m_2(i-1)+j-1}, y).$$

Let I_2 be the set of all such instructions. (Observe again that $x, y \in \Sigma^k$, and both I_1 and I_2 are finite sets of instructions). Then $M = (\Sigma, I_1 \cup I_2)$ is the required automaton.

We can easily generalize Lemma 2.2 and Lemma 2.3 to fit to our case and prove that $L_C(M) = L(G') \cup \{\lambda\}$. We omit these obvious proofs and verify only that the resulting automaton M has the desired properties. The properties 1, 3 and 4 can be easily verified. The only interesting property is the property 2. Consider any instruction $(x, u \to \Delta^{m_0+m_2(i-1)+j-1}, y) \in I_2$, where $i \in \{1, 2, \dots, m\}$ and $j \in \{1, 2, \dots, m_2\}$. We know that $|x| = |y| = k$ and $|xuy| > c = U + m_1 - 1$. Therefore, $|u| \geq m_0 + m_1 + m_2m - 1$. Since $m_0 + m_2(i-1) + j - 1 \leq m_0 + m_2m - 1$, we immediately get that $|u| - |\Delta^{m_0+m_2(i-1)+j-1}| \geq m_1$.

The following lemma will be important later:

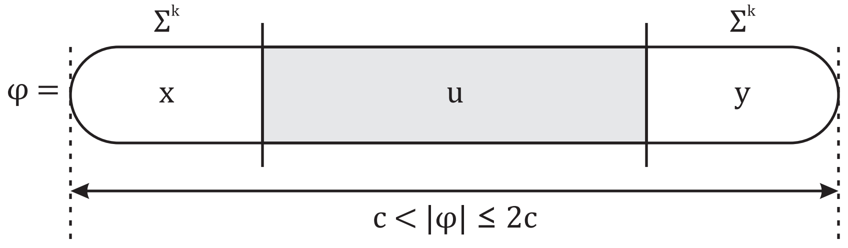


Figure 1: The illustration of the instruction $\phi = (x, u \to \Delta^{m_0+m_2(i-1)+j-1}, y)$.

Lemma 2.4. *For each $t \geq 1$, we can set the parameters m_1 and k in such a way, that for each instruction $(x, u \to \Delta^r, y) \in I_2$, there exists a subword $v \in \Sigma^*$ (not containing Δ) in the word u with the length $|v| \geq t$. Moreover, $m_1, k = \Theta(t)$. The parameters m_0 and m_2 can be chosen arbitrarily.*

Proof. Set $k := t$ and $m_1 := 2t + m_2m - 1$. Consider any instruction $(x, u \to \Delta^r, y) \in I_2$. If $u \in \Sigma^*$ then $v := u$ and we obtain $|v| = |u| > m_1 > t$. Suppose that there are some letters Δ in u. If we can find two consecutive continuous sequences of letters Δ in u, then at least k letters from Σ separate these two sequences. We can set v to be this separator. Suppose that there is only one continuous sequence of letters Δ, i.e. $u = w_1\Delta^s w_2$ for some $w_1, w_2 \in \Sigma^*$. We know that $|w_1| + s + |w_2| - r \geq m_1$. On the other hand, $|w_1| + s + |w_2| - r \leq |w_1| + |w_2| + m_2m - 1$, since $m_0 \leq r, s \leq m_0 + m_2m - 1$. Accordingly, $|w_1| + |w_2| + m_2m - 1 \geq m_1$. The longer of the words w_1, w_2 has the length at least $\frac{1}{2}(m_1 - m_2m + 1) = t$. $\square$

In the following, we would like to outline the basic idea behind the transformation of a k-Δ^*cl-RA M obtained from the previous construction into an equivalent Δcl-RA N. First suppose that we do this transformation in a trivial way, i.e. we transform each instruction $\phi = (x, u \to \Delta^r, y)$ of M, where $r > 1$ and $u = u_1 \dots u_s$, into the following set of so-called *partial instructions*:

$$\begin{aligned}
\phi_1 \quad &= (x, u_1 \to \Delta, u_2u_3 \dots u_s y),\\
\phi_2 \quad &= (x\Delta, u_2 \to \Delta, u_3u_4 \dots u_s y),\\
\dots \quad &\\
\phi_{r-1} \quad &= (x\Delta^{r-2}, u_{r-1} \to \Delta, u_r u_{r+1} \dots u_s y),\\
\phi_r \quad &= (x\Delta^{r-1}, u_r u_{r+1} \dots u_s \to \Delta, y).
\end{aligned}$$

Apparently, this technique gives us only the inclusion $L(M) \subseteq L(N)$. The other inclusion is not guaranteed. The problem is that we can find two different instructions ϕ and ψ in M such that they have different partial instructions ϕ_i and ψ_j applicable in the same context. One possible way how to avoid such situations is to introduce some new special instructions, which will encode some extra information into u by rewriting some letters with auxiliary Δ-symbols. By using Lemma 2.4 we can guarantee a long enough subword $v \in \Sigma^*$ in u, which we can use to encode this information. In the rest of this report we describe how to accomplish this task by using one specific coding. However, before we jump into the description of this coding, we introduce a new algorithmic viewpoint on clearing restarting automata, which somehow resembles interactive protocols or even Arthur-Merlin games from the complexity theory. This viewpoint will later simplify many complex constructions used in this report which would otherwise be very technical and difficult to understand.

3 Algorithmic Viewpoint

Let Σ be an input alphabet not containing the sentinels ¢ and \$. Consider two nondeterministic machines: the *querier* Q, the *solver* S, and the following *protocol*:

Meta-Algorithm 3.1. *Protocol describing the work of the querier Q and the solver S.*

Input: *Word $u \in \Sigma^*$.*

Description:

```
Let $w \leftarrow ¢ \cdot u \cdot \$$.
Repeat:
(1)  The querier $Q$ chooses a subword $x$ in $w$, i.e. $w = w_1 x w_2$.
(2)  If the solver $S$ accepts $x$ then Accept and halt.
(3)  If the solver $S$ answers $y$ on the input $x$ then set $w \leftarrow w_1 y w_2$.
```

The protocol (Q, S) accepts a word u if and only if there exists an accepting computation. Let $L(Q, S)$ denotes the language recognized by the protocol (Q, S):

$$L(Q,S) = \{ \; u \;\mid\; \text{protocol } (Q,S) \text{ accepts } u \; \}$$

Consider a class of queriers $\mathcal{Q}$ and a class of solvers $\mathcal{S}$. Then $\mathcal{L}(\mathcal{Q}, \mathcal{S})$ denotes the class of languages recognized by these queriers and solvers:

$$\mathcal{L}(\mathcal{Q},\mathcal{S}) = \{ \; L(Q,S) \;\mid\; Q \in \mathcal{Q}, S \in \mathcal{S} \; \}$$

In the following we will consider only the class of queriers $\mathcal{Q} = \{Q_1, Q_2, \ldots\}$, such that each querier $Q_K \in \mathcal{Q}$ for the given input word w chooses nondeterministically an arbitrary subword x of the word w of the length $|x| \leq K$.

The only constraint we put on the solver S is that it should preserve the sentinels ¢ and \$. In other words, the solver can neither erase these sentinels, nor create new ones. We are not interested in the time or space complexity of the solver S.

On the other hand, depending on the other constraints we put on the solvers we obtain different classes of solvers, and therefore also different classes of languages $\mathcal{L}(\mathcal{Q}, \mathcal{S})$. In the following we list several examples.

1. Consider the class of solvers $\mathcal{S}_{cl}$, such that each solver $S \in \mathcal{S}_{cl}$ works according to the following schema:

   (a) if the input word $x = ¢ \cdot \lambda \cdot \$$ then **Accept**,

(b) otherwise, either **Reject**, or return y, which can be obtained from x by deleting some substring of x (and preserving the sentinels ¢ a \$).

It is easy to see that $\mathcal{L}(\mathcal{Q}, \mathcal{S}_{cl}) = \mathcal{L}(\mathsf{cl\text{-}RA})$.

The inclusion $\mathcal{L}(\mathsf{cl\text{-}RA}) \subseteq \mathcal{L}(\mathcal{Q}, \mathcal{S}_{cl})$ is trivial. Suppose that $M = (\Sigma, I)$ is a k-$\mathsf{cl\text{-}RA}$. Let us define K to be the maximal width of instructions of M. The solver S will work according to the above schema in such a way that it will erase a substring from the given input word x only if there exists an instruction of the automaton M which allows such erasing. Apparently, $L(Q_K, S) = L(M)$.

For the proof of the other inclusion $\mathcal{L}(\mathcal{Q}, \mathcal{S}_{cl}) \subseteq \mathcal{L}(\mathsf{cl\text{-}RA})$ we use the fact the the querier only asks queries with the length bounded above by some constant K. Suppose that we have a protocol (Q_K, S), where the solver S works according to the above schema. There exist only finite many different queries the querier Q_K may ask. For each such query x we can describe the behavior of the solver S on the input x by using only the instruction of clearing restarting automata. If we unite all such instructions over all the queries the querier Q_K may ask then we get the required $\mathsf{cl\text{-}RA}$ M, such that $L(M) = L(Q_K, S)$.

2. The class $\mathcal{S}_{\Delta cl}$ consists of such solvers S that work according to the same schema as presented in 1 with the only exception: in the case 1b, the solver S can leave a mark $\Delta \notin \Sigma$ at the place of deleting. Apparently, $\mathcal{L}(\mathcal{Q}, \mathcal{S}_{\Delta cl}) = \mathcal{L}(\Delta\mathsf{cl\text{-}RA})$.

3. By analogy, consider the class $\mathcal{S}_{\Delta^* cl}$ which consists of solvers working according to the schema presented in 1 with the only exception: in the case 1b the solver S can leave a continuous segment Δ^r at the place of deleting, where $\Delta \notin \Sigma$ and r is bounded above by the length of the deleted substring. Not surprisingly, $\mathcal{L}(\mathcal{Q}, \mathcal{S}_{\Delta^* cl}) = \mathcal{L}(\Delta^*\mathsf{cl\text{-}RA})$.

4. The class $\mathcal{S}_{\mathsf{CFL}}$ consists of such solvers S that work according to the following schema:

   (a) if $x = ¢ \cdot N_1 \cdot \$$ then **Accept**,

   (b) if x contains ¢ or \$ then **Reject**,

   (c) if x contains neither ¢, nor \$, then either **Reject**, or return a nonterminal.

   We prove that $\mathcal{L}(\mathcal{Q}, \mathcal{S}_{\mathsf{CFL}}) = \mathsf{CFL}$.

   $\mathsf{CFL} \subseteq \mathcal{L}(\mathcal{Q}, \mathcal{S}_{\mathsf{CFL}})$: Let $G = (V_N, V_T, N_1, P)$ be a context-free grammar with $V_N = \{N_1, \ldots, N_m\}$, and $¢, \$ \notin V_N \cup V_T$. Let K be the length of the longest right-hand side of productions from P. The corresponding solver S will work according to the above schema in such a way, that in the case 4c S returns the nonterminal N_i only if $(N_i \to x) \in P$. Apparently, $L(Q_K, S) = L(G)$.

   $\mathcal{L}(\mathcal{Q}, \mathcal{S}_{\mathsf{CFL}}) \subseteq \mathsf{CFL}$: There exist only finite many different queries the querier Q_K may ask. For each such query x we can describe the behavior of the solver S on the input x by using context-free productions of the form $(N_i \to x)$. If we unite all such productions over all the queries the querier Q_K may ask then we get the required context-free grammar G, such that $L(G) = L(Q_K, S)$.

5. The class $\mathcal{S}_{\mathsf{CSL}}$ consists of such solvers S that work according to the same schema as presented in 4 with the only exception: in the case 4c, the solver S can rewrite any subword of the input word x to a nonterminal N_i. Analogously, $\mathcal{L}(\mathcal{Q}, \mathcal{S}_{\mathsf{CSL}}) = \mathsf{CSL}$.

In this way we can characterize also other language classes, such as pure languages [9] etc. All these classes have in common some kind of reduction analysis: we iteratively reduce the input word to the point in which we can decide whether the word belongs to the language or not. Moreover, all these reductions are local.

Also note, that if we restrict the class of queriers $\mathcal{Q} = \{Q_1, Q_2, \ldots\}$ in such a way that each querier $Q_K \in \mathcal{Q}$ can choose nondeterministically only a prefix or a suffix x of the given input word w with the length $|x| \leq K$, then the aforementioned classes will all collapse to the class of regular languages.

However, the most important contribution of these protocols is that we no more perceive any cl-RA ($\Delta\mathsf{cl}$-RA, $\Delta^*\mathsf{cl}$-RA, respectively) automaton only as a mere set of instructions, but rather as a nondeterministic machine S with an unbounded computational power working according to a particular schema. Instead of giving a list of instructions we rather describe the algorithm behind the corresponding solver. Now consider a k-$\Delta^*\mathsf{cl}$-RA whose construction was based on a given context-free grammar G as described in Section 2. Our goal is now to construct a solver $S \in \mathcal{S}_{cl}$ which will somehow imitate the work of the automaton M in such a way, that S will split one instruction of M into several steps. In the first phase S will encode into the tape (by using a special coding) the information about which instruction of M the solver S is going to imitate. In the second phase S will start building a continuous segment Δ^r. The encoded information will exactly describe the start and the end of this continuous segment and also which part of the input tape must be cleared in the end in order to complete the realization of the instruction of M. The coding will be designed in such a way that it will be possible to unambiguously interpret the content of the input tape at any given time.

4 Coding

Consider a finite nonempty alphabet Σ and $\Delta \notin \Sigma$. Our goal is to describe a mechanism which would enable us to encode any information to an arbitrary, sufficiently long word $w \in \Sigma^*$, only by replacing some letters of w by symbols Δ. Moreover, we require that it should be possible to recover the original word w at any time.

Theorem 4.1 (Coding 1)**.** *Let Σ be a finite nonempty alphabet and $\Delta \notin \Sigma$. Then there exist a positive integer B and a table T of triples (x, z, y), $xzy \in \Sigma^B$, $z \in \Sigma$, such that:*

1. *$\{xzy \mid (x, z, y) \in T\} = \Sigma^B$,*

2. *For each pair (x, y), $xy \in \Sigma^{B-1}$ there exists exactly one $z \in \Sigma : (x, z, y) \in T$.*

This theorem guarantees us that if we take any word $w \in \Sigma^B$ then there exists a factorization $w = xzy$, such that $(x, z, y) \in T$. Now if we replace the letter z by Δ, we do not lose any information, since we are able to recover the letter z from the context (x, y) by using the table T.

Proof. Let us set $B = |\Sigma|$ and define a bipartite graph $G = (U \cup V, E)$ as follows:

1. $U = \Sigma^B$,
2. $V = (\Sigma \cup \{\Delta\})^B \cap (\Sigma^* \cdot \Delta \cdot \Sigma^*)$,
3. $E = \{\{u, v\} \mid u \in U, v \in V, u = xzy, v = x\Delta y\}$.

There is an edge $\{u, v\} \in E$ between $u \in U$ and $v \in V$ if and only if v can be obtained from u by replacing one of its letters by the symbol Δ (see Figure 2).

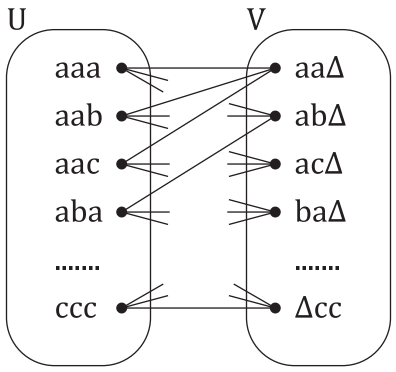


Figure 2: Bipartite graph $G = (U \cup V, E)$ for $\Sigma = \{a, b, c\}$.

It is easy to see, that:

1. $|U| = |\Sigma^B| = B^B$,
2. $|V| = B|\Sigma|^{B-1} = B^B$.

Moreover, the degree of each $u \in U$ in G is $B = |\Sigma|$ and the degree of each $v \in V$ in G is $|\Sigma|$. Therefore, G is a $|\Sigma|$-regular bipartite graph. By Hall's Theorem [3] there exists a perfect matching in G, which gives us the required table T. □

Example 4.1. *Consider $\Sigma = \{a, b\}$. The adjacency matrix of the corresponding bipartite graph G used in the proof of Coding 1 Theorem 4.1 is shown in Table 1.*

The highlighted perfect matching gives us the following bijection:

$$aa \leftrightarrow a\Delta, \quad ab \leftrightarrow \Delta b, \quad ba \leftrightarrow \Delta a, \quad bb \leftrightarrow b\Delta,$$

and the corresponding table $T = \{\ (a, a, \lambda),\ (\lambda, a, b),\ (\lambda, b, a),\ (b, b, \lambda)\ \}$.

Example 4.2. *For $\Sigma = \{a, b, c\}$ we only give the resulting bijection:*

	$a\Delta$	$b\Delta$	Δa	Δb
aa	$\underline{\mathbf{1}}$	0	1	0
ab	1	0	0	$\underline{\mathbf{1}}$
ba	0	1	$\underline{\mathbf{1}}$	0
bb	0	$\underline{\mathbf{1}}$	0	1

Table 1: Adjacency matrix for $\Sigma = \{a, b\}$.

$$\begin{array}{lllll}
aaa \leftrightarrow \Delta aa, & aab \leftrightarrow \Delta ab, & aac \leftrightarrow aa\Delta, & aba \leftrightarrow \Delta ba, & abb \leftrightarrow a\Delta b, \\
abc \leftrightarrow ab\Delta, & aca \leftrightarrow a\Delta a, & acb \leftrightarrow ac\Delta, & acc \leftrightarrow a\Delta c, & baa \leftrightarrow b\Delta a, \\
bab \leftrightarrow b\Delta b, & bac \leftrightarrow ba\Delta, & bba \leftrightarrow bb\Delta, & bbb \leftrightarrow \Delta bb, & bbc \leftrightarrow \Delta bc, \\
bca \leftrightarrow \Delta ca, & bcb \leftrightarrow bc\Delta, & bcc \leftrightarrow b\Delta c, & caa \leftrightarrow c\Delta a, & cab \leftrightarrow ca\Delta, \\
cac \leftrightarrow \Delta ac, & cba \leftrightarrow cb\Delta, & cbb \leftrightarrow c\Delta b, & cbc \leftrightarrow c\Delta c, & cca \leftrightarrow cc\Delta, \\
ccb \leftrightarrow \Delta cb, & ccc \leftrightarrow \Delta cc. & & &
\end{array}$$

Next consider the following sample word:

$$w = accbabccacaabbcabcbcacaa.$$

Let us factorize this word w into the groups of $B = 3$ letters:

$$w \;=\; acc \mid bab \mid cca \mid caa \mid bbc \mid abc \mid bca \mid caa.$$

Suppose that we want to encode the information $i = 11001010$ into the word w without losing any information. We can achieve this by marking *the groups of w by Δ that correspond to 1s in the information i. If we use the above bijection we will be able to recover the original word w. After encoding the information i we get the following word:*

$$w' \;=\; a\Delta c \mid b\Delta b \mid cca \mid caa \mid \Delta bc \mid abc \mid \Delta ca \mid caa.$$

By applying the encoding from Example 4.2 we can store an n-bit information in any word w of length at least nB. However, we need to "see" the whole word w in order to correctly define the groups of B letters.

The perfect matching of a regular bipartite graph G can be found effectively with respect to the size of the graph G. However, the table defining the bijection contains $|\Sigma|^{|\Sigma|}$ entries. It is an open problem, whether there is an algorithm realizing the bijection in a polynomial time with respect to $|\Sigma|$ without knowing the defining table.

As we can see, the coding introduced in Coding 1 Theorem 4.1 is *length-preserving.* It means, that if we encode an information into a word w, the length of the word remains preserved. The question is, whether we can find a *length-reducing* coding. The answer is: yes, trivially. Consider the following word:

$$w = ababaababbbaabaa.$$

First, we group together consecutive letters of w:

$$w = \underline{ab}\ \underline{ab}\ \underline{aa}\ \underline{ba}\ \underline{bb}\ \underline{ba}\ \underline{ab}\ \underline{aa}.$$

Now consider the word w as a word over the *pair* alphabet:

$$\underline{\Sigma} = \{\underline{aa}, \underline{ab}, \underline{ba}, \underline{bb}\}.$$

If we apply the length-preserving coding of Coding 1 Theorem 4.1 on the pair alphabet $\underline{\Sigma}$, we automatically get a length-reducing coding over the original alphabet Σ. This is because the rewriting of one letter from the pair alphabet $\underline{\Sigma}$ by Δ is equivalent to the rewriting of two letters from the original alphabet Σ by Δ. However, in this case $B = 2|\underline{\Sigma}| = 2|\Sigma|^2$. For example, for $\Sigma = \{a, b\}$ we need groups of $B = 2|\Sigma|^2 = 8$ letters. Can we do better? The answer is: yes, but not too much.

Theorem 4.2 (κ-Reducing Coding 1)**.** *Let κ be a nonnegative integer, Σ a finite nonempty alphabet and $\Delta \notin \Sigma$. Then there exist a positive integer B and a table T of triples (x, z, y), $xzy \in \Sigma^B$, $z \in \Sigma^{\kappa+1}$, such that:*

1. *$\{xzy \mid (x, z, y) \in T\} = \Sigma^B$,*
2. *For each pair (x, y), $xy \in \Sigma^{B-\kappa-1}$ there exists exactly one $z \in \Sigma^{\kappa+1} : (x, z, y) \in T$.*

Proof. Again, consider a bipartite graph $G = (U \cup V, E)$, where:

1. $U = \Sigma^B$,
2. $V = (\Sigma \cup \{\Delta\})^{B-\kappa} \cap (\Sigma^* \cdot \Delta \cdot \Sigma^*)$,
3. $E = \{\{u, v\} \mid u \in U, v \in V, u = xzy, v = x\Delta y\}$.

There is an edge $\{u, v\} \in E$ between $u \in U$ and $v \in V$ if and only if v can be obtained from u by replacing one of its subwords $z \in \Sigma^{\kappa+1}$ by Δ. If we define $B = |\Sigma|^{\kappa+1} + \kappa$ then we get the equality $|U| = |V|$, because:

1. $|U| = |\Sigma^B| = |\Sigma|^B$,
2. $|V| = (B - \kappa)|\Sigma|^{B-\kappa-1} = |\Sigma|^{\kappa+1}|\Sigma|^{B-\kappa-1} = |\Sigma|^B$.

Moreover, the degree of each $u \in U$ in G is $B - \kappa = |\Sigma|^{\kappa+1}$ and the degree of each $v \in V$ in G is $|\Sigma|^{\kappa+1}$. Therefore, G is a $|\Sigma|^{\kappa+1}$-regular bipartite graph. By Hall's Theorem [3] there exists a perfect matching in G, which gives us the required table T. □

For example, for $\Sigma = \{a, b\}$ the 1-Reducing Coding 1 gives us $B = |\Sigma|^{\kappa+1} + \kappa = 5$, which is an improvement to $B = 8$. However, this is not asymptotically better.

The obvious problem with Coding 1 is that we need to "see" the whole word w in order to correctly define the groups of B letters. One possible way how to avoid this problem is to find a coding that is not dependent on any specific factorization of the word to the groups of letters. Our goal is to be able to recover the original letters hidden by the symbols Δ only with the knowledge of local contexts of these symbols Δ.

Theorem 4.3 (Coding 2)**.** *Let $\Sigma = \{a, b\}$. There exist positive integers B, K, and a table T of triples (x, z, y), $x, y \in \Sigma^K$, $z \in \Sigma$, such that:*

1. *For each context $(x, y) \in \Sigma^K \times \Sigma^K$ there exists at most one triple $(x, z, y) \in T$.*
2. *For each word $w \in \Sigma^B$ there exists at least one triple $(x, z, y) \in T$ such that xzy is a subword of w.*

Apparently, we can replace the letter z by Δ in the subword xzy of w without losing any information, because we can recover z from the context (x, y).

Proof. Let us set $B = 8$, $K = 2$, and

$$\begin{array}{llllll} T = \{ & (aa, a, aa), & (ab, a, aa), & (ba, b, aa), & (bb, a, aa), & \\ & (aa, a, ab), & (ab, a, ab), & (ba, b, ab), & (bb, a, ab), & \\ & (aa, b, ba), & (ab, a, ba), & (ba, b, ba), & (bb, b, ba), & \\ & (aa, b, bb), & (ab, a, bb), & (ba, b, bb), & (bb, b, bb) & \} . \end{array}$$

The condition 1 obviously holds. The condition 2 is verified in Table 2.

$\underline{\text{aaaaa}}$???	$\underline{\text{aaaab}}$???	aa$\underline{\text{abaaa}}$?	aa$\underline{\text{abaab}}$?
aa$\underline{\text{ababa}}$?	aa$\underline{\text{ababb}}$?	a$\underline{\text{aabba}}$??	a$\underline{\text{aabbb}}$??
a$\underline{\text{abaaa}}$??	a$\underline{\text{abaab}}$??	a$\underline{\text{ababa}}$??	a$\underline{\text{ababb}}$??
$\underline{\text{aabba}}$???	$\underline{\text{aabbb}}$???	$\underline{\text{abaaa}}$???	$\underline{\text{abaab}}$???
$\underline{\text{ababa}}$???	$\underline{\text{ababb}}$???	a$\underline{\text{bbaaa}}$??	a$\underline{\text{bbaab}}$??
ab$\underline{\text{babaa}}$?	ab$\underline{\text{babab}}$?	ab$\underline{\text{babba}}$?	ab$\underline{\text{babbb}}$?
ab$\underline{\text{bbaaa}}$?	ab$\underline{\text{bbaab}}$?	abb$\underline{\text{babaa}}$	abb$\underline{\text{babab}}$
abb$\underline{\text{babba}}$	abb$\underline{\text{babbb}}$	a$\underline{\text{bbbba}}$??	a$\underline{\text{bbbbb}}$??
b$\underline{\text{aaaaa}}$??	b$\underline{\text{aaaab}}$??	baa$\underline{\text{abaaa}}$	baa$\underline{\text{abaab}}$
baa$\underline{\text{ababa}}$	baa$\underline{\text{ababb}}$	ba$\underline{\text{aabba}}$?	ba$\underline{\text{aabbb}}$?
ba$\underline{\text{abaaa}}$?	ba$\underline{\text{abaab}}$?	ba$\underline{\text{ababa}}$?	ba$\underline{\text{ababb}}$?
b$\underline{\text{aabba}}$??	b$\underline{\text{aabbb}}$??	$\underline{\text{babaa}}$???	$\underline{\text{babab}}$???
$\underline{\text{babba}}$???	$\underline{\text{babbb}}$???	$\underline{\text{bbaaa}}$???	$\underline{\text{bbaab}}$???
b$\underline{\text{babaa}}$??	b$\underline{\text{babab}}$??	b$\underline{\text{babba}}$??	b$\underline{\text{babbb}}$??
b$\underline{\text{bbaaa}}$??	b$\underline{\text{bbaab}}$??	bb$\underline{\text{babaa}}$?	bb$\underline{\text{babab}}$?
bb$\underline{\text{babba}}$?	bb$\underline{\text{babbb}}$?	$\underline{\text{bbbba}}$???	$\underline{\text{bbbbb}}$???

Table 2: All possibilities for $w \in \Sigma^K$.

Note that T can be shortly described as $T = \{(xx, y, y?)\} \cup \{(xy, x, ??) \mid x \neq y\}$, where $x, y \in \{a, b\}$, and the symbol ? represents an arbitrary letter. The intuition behind the condition 2 is the following: consider a sufficiently long word $w \in \Sigma^*$. There are only two cases: either each (internal) letter in w is doubled, i.e. each (internal) letter x in w has a neighbor x, or there exists an (internal) letter x in w which does not have a neighbor x. In the first case the pattern $xxyy$ will occur, and in the second case the pattern xyx, where $x \neq y$, will occur. $\square$

The obvious advantage of Coding 2 is that we do not need to factorize the input word into the groups in order to decode Δ symbols. However, it is an open problem, whether Coding 2 works also for larger alphabets. If it does, we could use the pairing argument to prove the length-reducing version of Coding 2.

Since we can prove Coding 2 Theorem 4.3 only for two-letter alphabets, we will use only Coding 1 in our algorithms. For simplicity we will use only the length-preserving version of Coding 1. In the end of this report we will give an argument explaining that it is also possible to use the length-reducing version of Coding 1 in our algorithms.

As we have already mentioned before, the problem with Coding 1 is that the solver $S \in \mathcal{S}_{\Delta cl}$ may not be able to decode Δ symbols in the input word w, since w can often be only a small part of the original input tape. If the word w does not start with the left sentinel ¢ then we are not able to correctly factorize w into the groups of B letters, and thus we are not able to interpret Δ symbols occurring in the word w.

Fortunately, there exists a simple trick how to avoid this problem. In order to correctly factorize the input word w into the groups of B letters we need only some "fixed point", which exactly defines the starting position of the first group of the correct factorization. The left sentinel ¢ is one example of such fixed point. Thus in the first phase we distribute such fixed points throughout the whole input tape starting at the left sentinel ¢. The distances between two consecutive fixed points will be approximately constant. We illustrate this idea on the following simplified example. Suppose that we have the following word:

$$w = abacc\Delta bacbbacacbcbaacbcbacbacabab$$

The symbol Δ in w represents our fixed point and defines the following factorization:

$$w = abacc\boldsymbol{\Delta} \mid bac \mid bba \mid cac \mid bcb \mid aac \mid bcb \mid acb \mid aca \mid bab$$

We place the next fixed point into the 9th group to the right from the highlighted fixed point $\boldsymbol{\Delta}$ (by using the bijection from Example 4.2):

$$w' = abacc\Delta \mid bac \mid bba \mid cac \mid bcb \mid aac \mid bcb \mid acb \mid aca \mid b\Delta b$$

As you can see, the number of letters between two consecutive fixed points is either 3×8, or $3 \times 8 + 1$, or $3 \times 8 + 2$. We place another fixed point whenever the input word w is of the form $w \in ¢ \cdot \Sigma^{\geq 3\times 9}$ or $w \in \Sigma^* \cdot \Delta \cdot \Sigma^{\geq 3\times 9}$.

5 Idea of the Algorithm

In this section we describe the algorithm behind the solver $S \in \mathcal{S}_{\Delta cl}$, which imitates the work of the k-Δ^*cl-RA M constructed according to a given context-free grammar in Chomsky normal form (as described in Section 2). The automaton M works in a bottom-up manner. If the automaton recognizes that some subword w of the input tape can be derived from a nonterminal N_i, then it can replace the inner part of this subword w by the code Δ^r, where $r = m_0 + m_2(i-1) + j - 1$ for some $j \in \{1, 2, \ldots, m_2\}$, leaving the first k letters and the last k letters of w as separators. The segment Δ^r together with its separators represents a code for the nonterminal N_i. These separators have a nice and useful property: if one changes some letters in these separators, the acceptance of the whole word on the input tape remains unchanged.

Our solver S is not obliged to preserve the representation used by the automaton M. Moreover, because of some technical reasons, we will not represent the nonterminal N_i by using a continuous segment Δ^r. Instead, we will use the segment $\Delta x \Delta^{r-4} y \Delta$, where $x, y \in \Sigma$ are the so-called "holes". These holes are useful in the sense that they can unambiguously identify the start and the end of the segment $\Delta x \Delta^{r-4} y \Delta$ inside any word marked by other symbols Δ. For example, in the following word:

$$a\Delta cca\Delta\underline{\Delta b\Delta^{r-4}a\Delta}aab$$

we have underlined such a segment with holes. It will be clear later why we need this representation. However, for the simplicity of verbalization we will often use the following convention: whenever we talk about a *continuous segment* Δ^r *with holes*, we will always mean the segment $\Delta x \Delta^{r-4} y \Delta$, where $x, y \in \Sigma$ represent the holes.

In the following we introduce another conventions which will be used later in the description of the resulting algorithm. As we have already explained, if we want to encode the information into some word $w \in \Sigma^*$, we need to know the factorization of this word w into the groups of B letters (see Section 4). This factorization (once defined) cannot be changed in the course of the algorithm. Otherwise, we could misinterpret the symbols Δ occurring in the word w. In order to correctly factorize the input word we need to find the so-called *fixed point*. We recognize three types of fixed points:

1. *Fixed point* ¢: In the word $¢ \cdot w$, where $w \in \Gamma^*$, the fixed point ¢ defines the following groups:
$$¢ \mid w_1 \mid w_2 \mid w_3 \mid \ldots,$$
where $w = w_1 w_2 w_3 \ldots$, and $|w_1| = |w_2| = |w_3| = \ldots = B$. In other words, the start of the input tape is a fixed point.

2. *Fixed point* Δ^r *with holes*: In the word uw, where u is the segment Δ^r with holes, $r = m_0 + m_2(i-1) + j - 1$ for some $j \in \{1, 2, \ldots, m_2\}$, the fixed point u defines the following groups:
$$w_0 \mid w_1 \mid w_2 \mid w_3 \mid \ldots,$$
where $uw = w_0 w_1 w_2 \ldots$, $|w_0| = m_0 + m_2(i-1)$, and $|w_1| = |w_2| = |w_3| = \ldots = B$. In other words, the segment Δ^r with holes is a fixed point. Note that $|w_0| \leq r$. The reason why we factorize the word uw in this way will be explained later.

3. *Fixed point* $\mathbf{u\Delta v\Delta}$: In the word $u\Delta v\Delta w$, where $u \in \Sigma^{2B}$, $v \in \Sigma^{\leq 2B-2}$, $w \in \Sigma \cdot \Gamma^*$, the fixed point $\mathbf{u\Delta v\Delta}$ defines the following groups:
$$u\Delta v\Delta \mid w_1 \mid w_2 \mid w_3 \mid \ldots,$$
where $w = w_1 w_2 w_3 \ldots$, and $|w_1| = |w_2| = |w_3| = \ldots = B$. In other words, two symbols Δ, which are close to each other, represent a fixed point.

To prevent the accidental creation of fixed points of the type $\mathbf{u}\Delta\mathbf{v}\Delta$ we introduce the following convention. We do not encode the information straight into the groups of length B, but rather to the so-called *units*. A unit is defined as three consecutive groups of length B. If we want to mark a unit in order to encode one bit of information we always mark the middle group of the unit. The first and the third group of the unit serves as separators. Thanks to these separators any two consecutive symbols Δ in any two neighboring units are separated by at least $2B$ letters from Σ. We illustrate this idea on the following example. Consider $\Sigma = \{a, b, c\}$ and the following word:

$$abc\Delta^r accbabccacaabbcabcbcccaa,$$

where $r = m_0 + m_2(i-1)$. The fixed point Δ^r (with holes) defines the following groups:

$$abc\Delta^r \mid\mid acc \mid \mathbf{bab} \mid cca \mid\mid caa \mid \mathbf{bbc} \mid abc \mid\mid bcc \mid caa,$$

where the units are separated by two lines (||). The middle groups of the units are in bold.

Note that if we mark two consecutive units, we never get the fixed point of the type $\mathbf{u}\Delta\mathbf{v}\Delta$. However, we can always obtain such a fixed point by marking two consecutive groups, provided that there are at least two unmarked groups preceding the first marked group. We use this observation in the first phase of the algorithm, in which we distribute the fixed points of this type throughout the whole input tape.

In the following we introduce the term *working area*. Consider an input tape with all its fixed points. If we erase these fixed points, the input tape will break into the segments, which we refer to as the working areas (see Figure 3).

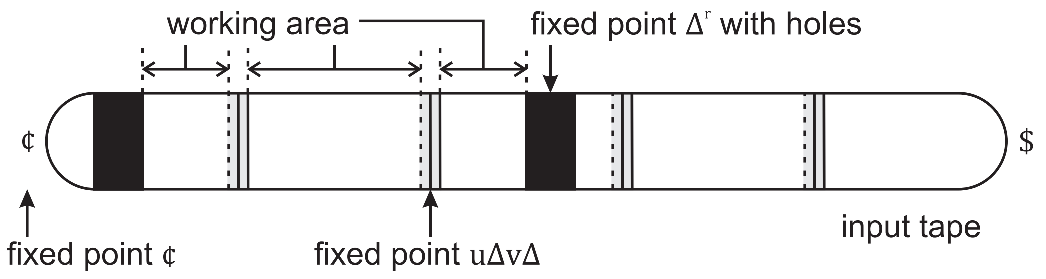


Figure 3: The segmentation of the input tape into the working areas.

Fixed points exactly define the factorization into the groups of length B and thus they also define the corresponding units inside these working areas. The term *working space* refers to the longest subword of the working area containing only the whole units that can be used to encode some information (see Figure 4).

Since we encode the information only into the middle groups of the units, no fixed points can be created in this way. However, there is one special exception. During the computation the algorithm gets into the state when it starts to create a new continuous segment Δ^r (with holes) somewhere inside the working space. This segment will (after completing) represent a code for a nonterminal. By definition, the completed segment Δ^r

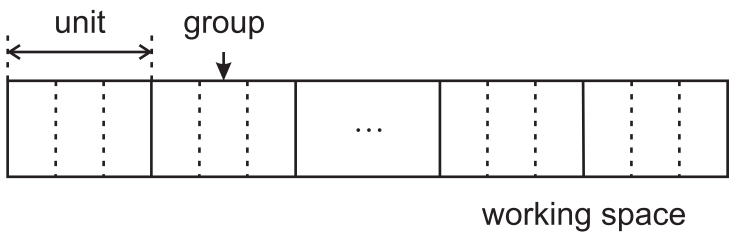


Figure 4: Working space divided into the units.

(with holes) represents also a new fixed point. This could be a problem, since the process of the realization of the instruction of the automaton M is not yet completed at this stage. In order to finish this process we must also clear the remaining letters on the both sides of the newly created segment Δ^r (with holes). The problem is, that a new fixed point could break the correctness of the previously defined factorization into the groups of length B. Therefore, we introduce some new terms that will enable us to solve this problem.

The working area is called *empty* if it does not contain a subword $u\Delta v$, where $u, v \in \Sigma^{4L}$. Otherwise we call the working area *reserved*. Next we will distinguish between *valid* and *invalid* fixed points. The fixed points of the types ¢ and $\mathbf{u}\Delta\mathbf{v}\Delta$ are always valid. The fixed point of the type Δ^r (with holes) is valid only if there exists at least one empty working area adjacent to this fixed point. Otherwise, the fixed point Δ^r (with holes) is invalid. Thus, if we want to make the fixed point Δ^r (with holes) invalid, we only need to place this fixed point between two marked units.

Now we are ready to give the details of the algorithm imitating the work of the k-Δ^*cl-RA M (see Meta-Algorithm 5.1). We explain the individual steps of this algorithm later.

Meta-Algorithm 5.1. *The algorithm describing the work of the solver $S \in \mathcal{S}_{\Delta cl}$, which imitates the work of the k-Δ^*cl-RA $M = (\Sigma, \Gamma, I)$.*

Input: *Word $w \in \{\lambda, ¢\} \cdot \Gamma^* \cdot \{\lambda, \$\}$.*

Description:

1. Find all fixed points inside w. Determine which of them are valid.
2. Let $w = z_0 o_0 z_1 o_1 \dots z_d o_d$, where:
a) z_0 is a fixed point (it may not be possible to decide if it is valid),
b) $z_1, \dots, z_d$ are the (decidable) valid fixed points of w,
c) $o_0, \dots, o_d$ are the corresponding working areas in w.
The working area o_d may contain the right sentinel \$.
*If it is not possible to factorize w in this way, especially if w does not start with the fixed point, then **Reject**.*
3. Identify the groups of length B and the corresponding units in the working areas $o_1, \dots, o_d$. If z_0 is a valid fixed point then take also the working area o_0 into consideration.
4. If the working area $o_d \in \Sigma^{\geq \mathsf{const}} \cdot \{\lambda, \$\}$ (where const will be specified

later), then add another fixed point of the type $\mathbf{u}\Delta\mathbf{v}\Delta$ *into this working area and* ***Halt****.*
5. Recover all symbols Δ *in the word* w *that can be recovered.*
We use the notation $\bar{u}$ *for the recovered word* u*.*
6. If $\bar{w} = ¢ \cdot \bar{u} \cdot \$$ *and* $(¢, \bar{u} \to \lambda, \$) \in I$*, then* ***Accept****.*
7. Otherwise, let $\phi = (x, u \to \Delta^r, y)$ *be the instruction of* M*, which can be applied in the subword* $z_\alpha o_\alpha \ldots z_\beta o_\beta$*, where both* z_α *and* z_β *are valid fixed points. More precisely, either there exists a reserved working area* o_γ $(\alpha \leq \gamma \leq \beta)$*, in which it is exactly encoded, which instruction* ϕ *is being realized in this area, or there is no such working area and* xuy *is a subword of the word* $\bar{z}_\alpha \bar{o}_\alpha \ldots \bar{z}_\beta \bar{o}_\beta$ *(Even if there was a reserved working area* o_γ*, then the encoding process in this area might not be finished, which would automatically imply that all its* Δ *symbols can be recovered). In the case that all the working areas* $o_\alpha, \ldots, o_\beta$ *are empty, we only need to choose one of them. We know that in the word* u *there exists a sufficiently long subword* $v \in \Sigma^*$ *in which we can encode the information. Suppose that* v *covers the working area* o_γ*, where* $\alpha \leq \gamma \leq \beta$*. Then we choose* o_γ *if both of the following two conditions hold:*
a) Either $\gamma = \alpha = 0$ *and* $z_0 = ¢$*, or* $\gamma > \alpha \geq 0$*,*
b) Either $\gamma = \beta = d$ *and* o_d *ends with the sentinel* $\$$*, or* $\gamma < \beta \leq d$*.*
If there is no such instruction then ***Reject****.*
8. Execute one step of the realization of the instruction ϕ *and* ***Halt****.*

Step 1 is clear. Fixed points are exactly defined nonoverlapping segments (we suppose that $m_0 \geq 8$). If we delete these segments the input word w will break into the areas. These areas enable us to determine which fixed points are valid and which are not. The only exception is the first fixed point z_0. If z_0 is of the type Δ^r (with holes) then we may not be able to decide if z_0 is a valid fixed point, since we do not see to the left of z_0. We always suppose that the input word w starts with a fixed point (otherwise, we reject). Valid fixed points exactly define both the factorization $w = z_0 o_0 z_1 o_1 \ldots z_d o_d$ in Step 2, and the groups of length B and the corresponding units in Step 3. Thanks to these factorizations we are able to recover the symbols Δ occurring in the word w. If u is a subword of w then we denote by $\bar{u}$ the word u with all possible symbols Δ occurring in u recovered. We may not be able to recover all the symbols Δ in u. We will describe later when this happens. During the recovering process we also fill the holes of the segments Δ^r (with holes) by the symbols Δ. This is because the original automaton M uses as the code for the nonterminals the full segments Δ^r without holes.

Step 4 is important in the first phase of the algorithm when we distribute the fixed points of the type $\mathbf{u}\Delta\mathbf{v}\Delta$ throughout the whole input tape. As we have already mentioned, we create the fixed point of the type $\mathbf{u}\Delta\mathbf{v}\Delta$ by marking two consecutive groups by the symbols Δ. Therefore, it is necessary to split Step 4 into two smaller steps – each step for marking one group. In the case that it is not necessary to create another fixed point of the type $\mathbf{u}\Delta\mathbf{v}\Delta$, we continue with Step 5.

In Step 6 we cover the instructions from I_1 of the automaton M. This Step can be realized only in the case when we see the whole input tape (i.e. $z_0 = ¢$ and o_d ends with the symbol \$). This is the only place of the algorithm in which the solver can actually accept the input word.

The most interesting steps are Step 7 and Step 8. In Step 7 we first choose nondeterministically the instruction ϕ from I_2, which can be applied in the word $\bar{u}$. Since it is not possible for the solver S to realize this instruction in a single step, we need to choose a working area o_γ in which we will unfold the whole process of realizing this instruction ϕ into many individual steps. The conditions in Step 7 will guarantee us that the neighboring areas with the area o_γ will be empty (if they exist). The reason for this is that we do not want to unintentionally invalidate some other fixed points in Step 8. It is also possible that there already exists a reserved working area o_γ in the input word w. In that case we only continue in the process of the realization of the instruction encoded in this reserved working area. Step 8 is described in detail in the following Section 6.

6 Realization of single Instruction ϕ

In this section we describe Step 8 of Meta-Algorithm 5.1 in detail. Suppose that we want to imitate the instruction $\phi = (x, u \to \Delta^r, y)$ of the automaton M, where $x, y \in \Sigma^k$, $r = m_0 + m_2(i-1) + (j-1)$, $1 \le i \le m$, $1 \le j \le m_2$. The number i is fixed for this instruction. However, we can choose $j \in \{1, 2, \ldots, m_2\}$ arbitrarily. We will explain later which value we have to choose for j. Suppose that the instruction ϕ can be applied in the subword $z_\alpha o_\alpha \ldots z_\beta o_\beta$ of the input word w, where both z_α and z_β are valid fixed points. We unfold the process of the realization of the instruction ϕ into several individual steps. All these steps will be executed in the working area o_γ, where $\alpha \le \gamma \le \beta$. Suppose that the neighboring working areas with o_γ are empty (if they exist). Let p_γ be the working space corresponding to the working area o_γ, and $D_1, \ldots, D_h$ be all its units (see Figure 5).

Meta-Algorithm 6.1 describes the realization of the instruction ϕ in detail. The used variables and constants will be defined later. The interpretation of the working space p_γ is illustrated in Figure 6.

Meta-Algorithm 6.1. *The algorithm realizing one step of the instruction ϕ.*

> **Input:** *Word $w \in \{\lambda, ¢\} \cdot \Gamma^* \cdot \{\lambda, \$\}$, working area o_γ, [instruction ϕ].*
>
> **Description:** *From the following steps execute the first step, which was not yet executed and **Halt**. If you encounter a conflicting step, i.e. a step which is either not possible to execute or which does not correspond to the information already encoded in the working area o_γ, then **Reject**.*
> *1. Mark the unit D_2. The corresponding Δ is called the reference point. Let* left *be the position of the first letter of u relative to the reference point,* right *be the position of the last letter of u relative to the reference point. Compute j and $s = m_0 + m_2(i-1) + (j-1)$. If the instruction ϕ is not given at the input, compute these values from the information encoded in*

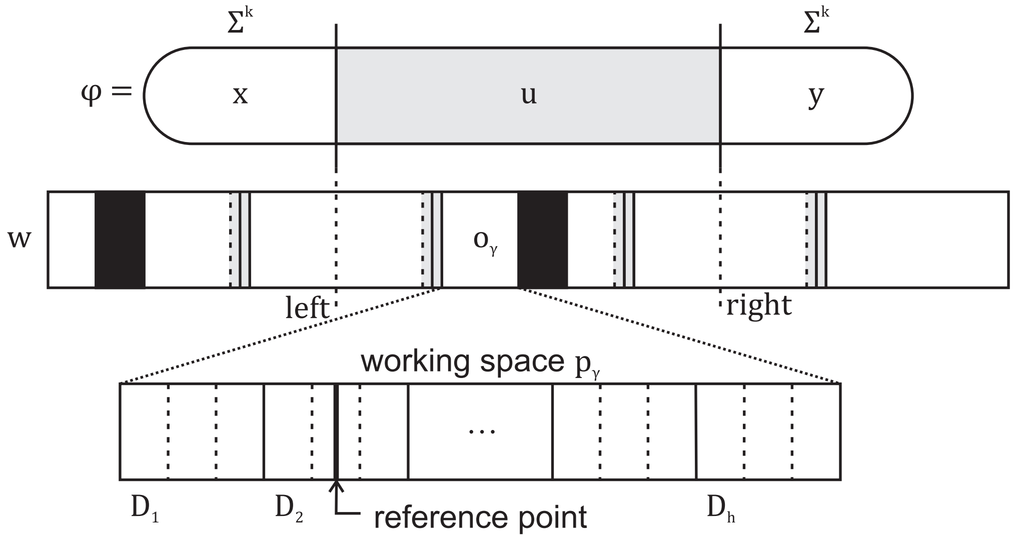
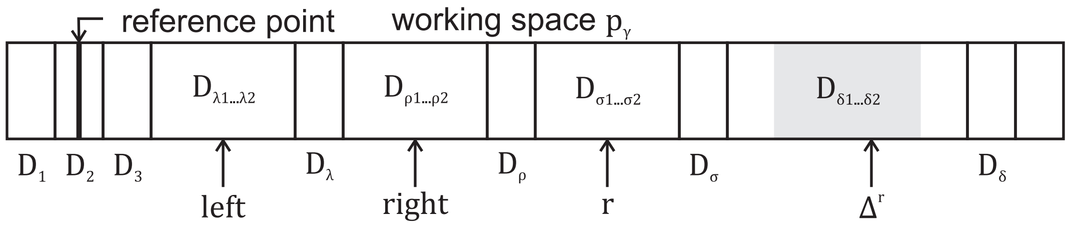


Figure 5: The realization of the instruction ϕ.



reference point
working space p_γ
$D_{\lambda 1 \ldots \lambda 2}$
$D_{\rho 1 \ldots \rho 2}$
$D_{\sigma 1 \ldots \sigma 2}$
$D_{\delta 1 \ldots \delta 2}$
D_1
D_2
D_3
D_λ
D_ρ
D_σ
D_δ
left
right
r
Δ^r

Figure 6: The interpretation of the working space p_γ.

*the corresponding units. If it is not possible, i.e. the units D_λ, D_ρ, D_σ, D_δ are not marked yet, then **Reject**.*
2. Encode left *into the units $D_{\lambda_1}, \ldots, D_{\lambda_2}$. Mark the unit D_λ.*
3. Encode right *into the units $D_{\rho_1}, \ldots, D_{\rho_2}$. Mark the unit D_ρ.*
4. Encode s into the units $D_{\sigma_1}, \ldots, D_{\sigma_2}$. Mark the unit D_σ.
5. Mark the unit D_δ.
6. Encode the segment Δ^r (with holes) into the units $D_{\delta_1}, \ldots, D_{\delta_2}$.
7. Clear all letters from the end of the segment Δ^r (with holes) to the position right*.*
8. Clear all letters from the position left *to the beginning of the segment Δ^r (with holes).*

The aforementioned algorithm is designed in such a way, that at any time it is possible to determine unambiguously which steps were already executed and which were not. Each step of the algorithm is consistent with some instruction ϕ of the simulated automaton M, therefore the solver cannot accept more than M. We will show later that it is possible to define the parameters and constants used in the solver in such a way, that each instruction of M can be simulated by the solver. Moreover, by using the length-reducing version of coding it is possible to obtain a length-reducing version of the solver which shortens the word in each step.

The reference point not only reserves the working area o_γ (the units D_1 and D_3 must remain empty), but it also exactly defines the relative positions of the first and the last letter of the word u. The number left is defined as the number of letters of the shortest prefix of u containing the reference point. By analogy, the number right is defined as the number of letters of the smallest suffix of u containing the reference point.

Steps 1 to 5 are nondestructive in the sense that we are able to recover all the symbols Δ created by these steps. However, when we start to create the continuous segment Δ^r (with holes) we lose the ability to recover all the symbols Δ in the working area o_γ. Therefore, before we start to create this segment, we need to have all the information necessary to complete the realization of the instruction ϕ already encoded in the working area o_γ. We use the standard binary representation for the numbers and we always encode the bits from the lowest significant bit to the most significant bit in the direction from the left to the right. For all the numbers left, right and r we use a fixed number of bits. Hence it is easy to recognize markings in units D_λ, D_ρ, D_σ and D_δ.

If the unit D_δ is marked then we are in the phase of creating the segment Δ^r (with holes), i.e. Step 6. We build this segment step by step from the left to the right by replacing the letters in the subword $D_{\delta_1}, \ldots, D_{\delta_2}$ by the symbols Δ. After the completion of this segment we move to the next Step 7. Note that before we execute Step 7 the newly created segment Δ^r (with holes) is not a valid fixed point, since this segment is placed between two marked units (the reference point and the marked unit D_δ). After executing Step 7 this segment becomes a valid fixed point. This is because we clear the marked unit D_δ and all the areas neighboring with the working area o_γ are empty (if they exist). Therefore, we need to choose the parameter j in such a way, that the newly defined groups of length

B, defined by the newly created fixed point of the type Δ^r (with holes), are exactly the same groups as the original groups before executing Step 7. In Step 8 this problem does not arise, because the groups of length B are always defined relatively to the right of a fixed point.

We conclude this section by mentioning one specific problem concerning Step 7 and Step 8. The problem is, that the position left (right, respectively) can cross the fixed point of the type $\mathbf{u}\Delta\mathbf{v}\Delta$ (see Figure 7).

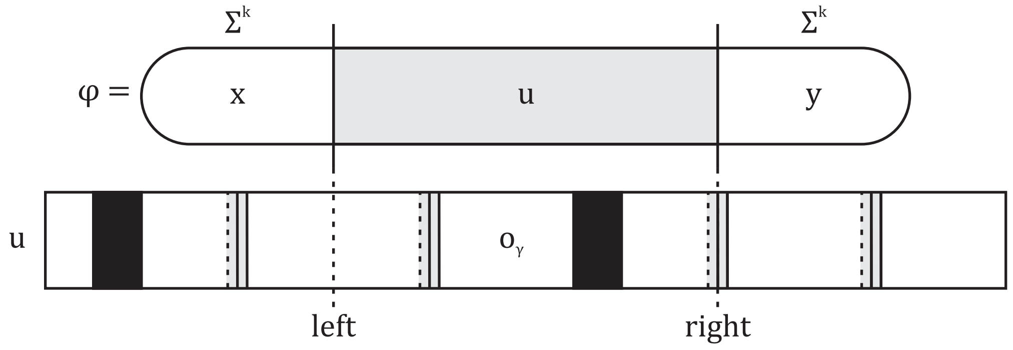


Figure 7: Problem with the fixed point of the type $\mathbf{u}\Delta\mathbf{v}\Delta$.

The positions left and right are fixed and we cannot change them (these positions are defined by the instruction ϕ). Fortunately, it does not matter if we damage the fixed point of the type $\mathbf{u}\Delta\mathbf{v}\Delta$. This is because the newly created fixed point of the type Δ^r (with holes) defines the groups of length B in exactly the same way as did the damaged fixed point. Moreover, thanks to the holes in the newly created fixed point we do not lose even the ability to exactly define the borders of this newly created fixed point. The only issue is that we lose the ability to recover the Δ symbol(s) of the damaged fixed point. Fortunately these Δ symbols are close enough to the newly created segment Δ^r (with holes), so they are situated in the separator corresponding to this segment. As we have already said, the letters in the separator can be set arbitrarily. Therefore, if we want to recover these symbols Δ of the damaged fixed point, we can use any letter we want.

Note that the position left (right, respectively) cannot cross the fixed point of the type Δ^r (with holes), because the word xuy covers always the whole code of the nonterminal (including its separators).

7 Choosing the Parameters

In this section we define all the parameters and constants used in the previous sections. Let Σ be a given alphabet and G be a context-free grammar with m nonterminals.

First we set $m_0 := 8$. Thanks to this setting, each segment Δ^r (with holes) will contain Δ^4 as a subword. This is how we recognize such segments.

Each group is $B = |\Sigma|$ letters long (since we use Coding 1) and each unit is (by definition) $3B$ letters long. The parameter m_2 defines the range for the parameter j in

Meta-Algorithm 6.1. Since we need this parameter j only to correctly adjust the groups of length B defined by the newly created fixed point, it suffices to choose $m_2 = 3B$.

The following computations will involve the parameter h, which represents the number of units necessary to allow the execution of Meta-Algorithm 6.1.

The numbers left and right are bounded above by the width of the instruction ϕ, i.e. by the constant $2c$, where $c = U + m_1 - 1$ and $U = m_0 + m_2m - 1 + 2k$. Therefore, to encode such a number we need at most $\lceil \log_2(2c) \rceil$ units. The number r is bounded above by $m_0 + m_2m$. Observe that this upper bound depends neither on m_1, nor on k. To encode r we need at most $\lceil \log_2(m_0 + m_2m) \rceil$ units. Finally, to create the segment Δ^r (with holes) we need at most r letters, i.e. at most $\lceil \frac{m_0+m_2m}{3B} \rceil$ units. Therefore, we need at most

$$h = O(m_0 + m_2m + \log(m_1 + 2k)).$$

By Lemma 2.4 $m_1, k = \Theta(t)$. Therefore, for any real $\alpha > 0$, there exists a big enough t such that αt is bigger than h. By using this technique we can guarantee, for any instruction $\phi = (x, u \to \Delta^r, y)$, a big enough subword $v \in \Sigma^{\geq t}$ of the word u. This subword v will guarantee us the existence of a long enough working area o_γ inside this subword v. In our case it is not sufficient only to say that $v \in \Sigma^{\geq t}$ is long enough, because in our algorithm this subword can be interrupted by fixed points of the type $\mathbf{u}\Delta\mathbf{v}\Delta$. The fact that $v \in \Sigma^*$ only means that we are able to recover the symbols Δ of these fixed points. However, if v is long enough then there will be many of these fixed points and therefore also many of the corresponding working areas. Therefore, it will always be possible to choose one such area o_γ with empty neighbors.

Let us set $\alpha = \frac{1}{6}\frac{1}{3B}$, and let t be the smallest positive integer such that for the corresponding parameters m_1 and k the following holds:

$$\frac{1}{6}\frac{1}{3B}t > h.$$

In the first phase of our algorithm let us distribute the fixed point throughout the input working tape in such a way that the corresponding working spaces between any two consecutive fixed points contain at least $2h$ units. Since t is bigger than the length of $6h$ units, it is easy to see that we will always be able to find a long enough working area o_γ inside the subword $v \in \Sigma^{\geq t}$.

Also note that during the course of our algorithm the length of every working area will be bounded above by $2h$ units. This is because the fact that during the course of our algorithm we only replace some segments by new fixed points, so it is not possible to enlarge any working area in this way.

Finally, the constant K of the querier must be long enough to enable us to see any instruction ϕ of M as a whole, plus some extra working spaces on both sides of this instruction. It is sufficient to set $K = O(c + h) = O(c)$.

8 Length-Reducing Algorithm

In is not difficult to modify Meta-Algorithms 5.1 and 6.1 in order to work with the length-reducing version of Coding 1 (see Section 4). We only need to redefine the length of the word $|\cdot|$ to a weighted sum of its letters, where the letters from Σ have the weight 1 and the symbol Δ has the weight equal to $\kappa + 1$, where κ is the reduction factor from κ-Reducing Coding 1 Theorem 4.2. Moreover, in order to create the segment Δ^r (with holes) we need now $(\kappa + 1)$ times more letters, i.e. (at most) $(\kappa + 1)(m_0 + m_2 m)$ letters. When we create the segment Δ^r (with holes) we always rewrite $\kappa + 1$ consecutive letters from Σ to one symbol Δ. It is easy to see that the constant κ can be easily incorporated into the previous consideration in Section 7.

9 Conclusion

$\Delta\mathsf{cl\text{-}RA}$ are very limited in their operations. They can in one step either completely delete a subword or they can delete a subword and simultaneously mark its position by a single symbol Δ. Surprisingly, they can accept any context-free language. We have designed a rather complicated coding of information used for encoding nonterminals during a bottom-up analysis. It would be interesting to find a simpler encoding, e.g. without the fixed points.

Another open problem is to characterize exactly the class of languages recognized by $\Delta\mathsf{cl\text{-}RA}$ and $\Delta^*\mathsf{cl\text{-}RA}$. Obviously, $\mathcal{L}(\Delta\mathsf{cl\text{-}RA}) \subseteq \mathcal{L}(\Delta^*\mathsf{cl\text{-}RA})$, but we do not know whether this inclusion is strict. Of course, $\Delta\mathsf{cl\text{-}RA}$ and $\Delta^*\mathsf{cl\text{-}RA}$ should be more precisely related to the stateless restarting automata from [7] and [8]. In contrast to the stateless restarting automata, $\Delta\mathsf{cl\text{-}RA}$ and $\Delta^*\mathsf{cl\text{-}RA}$ can use only single auxiliary symbol Δ, but on the other hand, they need not to be length reducing.

References

[1] P. Černo and F. Mráz. Clearing restarting automata. In H. Bordinh, R. Freund, M. Holzer, M. Kutrib, and F. Otto, editors, *Workshop on Non-Classical Models for Automata and Applications (NCMA)*, volume 256 of *books@ocg.at*, pages 77–90. Österreichisches Computer Gesellschaft, 2009.

[2] P. Černo and F. Mráz. Clearing restarting automata. *Fundamenta Informaticae*, 104(1):17–54, 2010.

[3] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(1):26–30, 1935.

[4] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata.* Addison-Wesley, Reading, 1969.

[5] P. Jančar, F. Mráz, M. Plátek, and J. Vogel. Restarting automata. In H. Reichel, editor, *FCT'95*, volume 965 of *LNCS*, pages 283–292, Dresden, Germany, August 1995. Springer.

[6] M. Kutrib, H. Messerschmidt, and F. Otto. On stateless deterministic restarting automata. In M. Nielsen, A. Kučera, P. B. Miltersen, C. Palamidessi, P. Tůma, and F. D. Valencia, editors, *SOFSEM*, volume 5404 of *LNCS*, pages 353–364. Springer, 2009.

[7] M. Kutrib, H. Messerschmidt, and F. Otto. On stateless deterministic restarting automata. *Acta Inf.*, 47:391–412, December 2010.

[8] M. Kutrib, H. Messerschmidt, and F. Otto. On stateless two pushdown automata and restarting automata. *International Journal of Foundations of Computer Science*, 21:781–798, 2010.

[9] H. Maurer, A. Salomaa, and D. Wood. Pure grammars. *Information and Control*, 44(1):47 – 72, 1980.

[10] F. Otto. Restarting automata. In Z. Ésik, C. Martín-Vide, and V. Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 269–303. Springer, Berlin, 2006.