Forgetting Automata and Context-Free Languages

Petr Jančar *

Department of Computer Science, University of Ostrava

Dvořákova 7, 701 03 Ostrava, Czech Republic

e-mail: jancar@osu.cz

František Mráz

AND

Martin Plátek

Department of Computer Science, Charles University

Malostranské nám. 25, 118 00 Praha 1, Czech Republic

e-mail: mraz,platek@kki.ms.mff.cuni.cz

Abstract

It is shown that context-free languages can be characterized by linear bounded automata with the following restriction: the head can either move right without rewriting or move left with erasing the current cell (i.e. rewriting it with a special, nonrewriteable, symbol). If, instead of erasing, we consider deleting (complete removing of the cell), the corresponding automata are less powerful.

1 Introduction

Our motivation comes from the area of analysis (grammar checking) of natural as well as programming languages. The used methods are often based on the idea of stepwise leaving out some parts of the given sentence (e.g. parts of correct configurations in the sense of [8]) which do not affect the syntactic (non)correctness. We illustrate it by the following simple example.

"The little boy ran quickly away",
"The ____ boy ran quickly away",
"The ____ boy ran _____ away".

*This author is supported by the Grant Agency of the Czech Republic, Grant No. 201/93/2123

It leads us to considering linear bounded automata with limited rewriting ability: any symbol can only be rewritten by a special symbol @ (meaning the emptiness of the relevant cell). We refer to the operation as *erasing*.

Another natural possibility is to consider the operation of *deleting*, which "cuts out" the cell (and pastes the two remaining parts of the tape together). In the example

"The little boy ran quickly away",
"The boy ran quickly away",
"The boy ran away".

To make deleting more natural, we use the model of list automata; a list automaton uses a (doubly linked) list of items (cells) rather than the usual tape.

In our example there is no difference between erasing and deleting. But in general, erasing is more powerful – it can simulate deleting and, in addition, the erased items (unlike the deleted ones) can still carry information.

Erasing automata were also considered by von Braunmühl and Verbeek in [1]. The motivation was different: they introduced a new storage medium – the finite-change tape – which is, from the complexity point of view, "something between time and space"; the erasing automaton is a special case then.

In [6] and [7], we showed (or sketched) some results concerning the power of nondeterministic forgetting – i.e. erasing and/or deleting – automata. We find most interesting the results related to the class of context-free languages, denoted by CFL. Here we give completed versions of them, with full proofs.

First we should mention that nondeterministic forgetting automata are strictly less powerful than deterministic linear bounded automata, i.e. they recognize a proper subclass of deterministic context-sensitive languages; it is shown in [5] using a Savitch-like simulation (similarly as in [1]) and the method of diagonalization.

As regards CFL, it is immediately clear that e.g. the non-context-free language $\{a^n b^n c^n \mid n \geq 0\}$ is recognizable by a deleting (and hence also erasing) automaton. On the other hand, von Braunmühl and Verbeek doubt in [1] that every context-free language is recognizable by some erasing automaton; they show that it is true if we allow an additional Turing tape of length $log(n)$ (n being the length of the input).

Nevertheless, we show here that the additional tape is not necessary. Erasing automata not only recognize (more than) all context-free languages but their subclass – erasing automata which can not move left without erasing – characterizes CFL precisely.

The situation with deleting automata is more unclear. We conjecture that there is a context-free language (cf. Section 5) which is not recognizable by any deleting automaton but we have not proved it. What we do show is an (even deterministic) context-free language which is not recognizable by any deleting automaton which moves left by deleting only. This result confirms that erasing is *strictly* more general (more powerful) than deleting. In addition, the proof uses a technique of pumping lemmas for "computation trees" (resembling those in [9]) which we find interesting on its own.

The paper is organized as follows. Section 2 contains definitions and the main results; the proofs are contained in Sections 3,4. In Section 5, some additional remarks are given.

2 Definitions and Results

A *forgetting automaton* F is formally a tuple $F = (Q, \Sigma, \delta, q_0, Q_A)$, where

Q is a finite set of *states*,
$q_0 \in Q$ is the *initial state*,
$Q_A \subseteq Q$ is the set of *accepting states*,
Σ is a finite set (*alphabet*) not containing ¢,\$,@ and
$\delta : Q \times (\Sigma \cup \{¢, \$, @\}) \to \mathcal{P}(Q \times OP)$ is the *transition function*, OP being the set $OP = \{MV_R, MV_L, ER_R, ER_L, DL_R, DL_L\}$.

($\mathcal{P}(A)$ denotes the set of subsets of A.)

F is supposed to have a control unit with the state set Q and one head moving on a linear (doubly linked) list of items (cells); each item contains a symbol from $\Sigma \cup \{¢, \$, @\}$. In the *initial configuration*, the control unit is in the initial state q_0, the list contains an input word $w \in \Sigma^*$ bounded by special sentinels ¢, \$ and the head scans the item with the left sentinel ¢.

F performs the computation according to δ, which can be viewed as a set of *instructions* in an obvious sense. $\delta(q, a) \ni (q', op)$, written also as $(q, a) \to (q', op)$, means that if the current state is q and the scanned symbol is a then F can change the state to q' and perform op which is one of the following six operations:

- MV_R, MV_L — moving the head one item to the right (left),
- ER_R, ER_L — erasing, i.e. rewriting the contents of the scanned item with the special symbol @ and moving the head one item to the right (left),
- DL_R, DL_L — deleting the item from the list moving the head one item to the right (left).

For any instruction I of the form $(q, a) \to (q', op)$, we suppose that $a = ¢$ implies $op = MV_R$ and $a = \$$ implies $op = MV_L$. If $op = MV_R$, we call I an MV_R–instruction; similarly for the other op's.

Notice that, in general, F is *nondeterministic* (more than one instruction can be applicable at the same time).

An input *word w is accepted by F* if there is a computation starting in the initial configuration corresponding to w, which reaches a configuration with the control unit being in one of the accepting states.

$L(F)$ denotes the language consisting of all words accepted by F; we say that F *recognizes* $L(F)$.

By (MV_R, ER)–languages we mean the languages recognized by (MV_R, ER)–automata, i.e. forgetting automata having MV_R–,ER_R– and ER_L–instructions only; for technical reasons, instructions of the type $(q, \$) \to (q', MV_L)$ are also allowed ($ being the right sentinel). The class of (MV_R, ER)–languages is denoted by $\mathcal{L}(MV_R, ER)$. Similarly we define (MV_R, DL)–languages (automata) and the class $\mathcal{L}(MV_R, DL)$.

The following theorems are proven in Sections 3,4 respectively.

Theorem I. *$\mathcal{L}(MV_R, ER)$ is equal to CFL.*

Theorem II. *$\mathcal{L}(MV_R, DL)$ is a proper subset of CFL.*

3 $\mathcal{L}(MV_R, ER)$ is equal to CFL

The inclusion $\mathcal{L}(MV_R, ER) \subseteq CFL$ is easier to show. We prove it by standard methods of the automata theory.

Proposition 3.1. *Any (MV_R, ER)–language is context-free.*

Proof. We show how an arbitrary (MV_R, ER)–automaton E can be simulated by a pushdown automaton M.

Since any ER_R–instruction can be simulated by an ER_L– and two MV_R–instructions, we can suppose that E only has MV_R– and ER_L–instructions, possibly with instructions of the type $(q, \$) \to (q', MV_L)$. We also suppose that E visits all items before accepting.

The crucial point is to realize that any segment of consecutive erased items carries bounded information only. Entering such an erased segment, say from the right end in state q, E can, after some number of steps within the erased segment, halt in some state q' or leave the segment to the right or to the left in some state q'' or cycle infinitely.

Such a behaviour for an erased segment can be given by a function

$$B : (\{left, right\} \times Q) \longrightarrow \mathcal{P}((\{left, right, halt\} \times Q) \cup \{cycle\})$$

with the obvious meaning.

By B_n we denote the behaviour of E for the erased segment of length n. We also use B_0 for technical reasons.

The set $\mathcal{B}$ of behaviours of E for segments of all lengths is finite. Moreover, given behaviours B,B' for some erased segments, the behaviour for concatenation of these segments, denoted by $Conc(B, B')$, is fully determined by B,B' without referring to the lengths of segments. In particular, given B_n, $B_{n+1} = Conc(B_n, B_1)$ can be determined without knowing n.

M will have some "variables" in its finite control unit which will store the following information: CSt stores the current state of E, $CSym$ the current symbol, B^L the behaviour of the (rightmost) erased segment on the left w.r.t. the head, and B^R the behaviour of the erased segment on the right (the segments can be empty).

The idea is that the left hand side of the working list of E will be stored in the pushdown of M, where the erased segments are replaced by the corresponding behaviours.

Due to the information stored in the control unit of M, a whole sequence of instructions of E when working in an erased segment can be simulated by one or two steps of M.

Consider the case when E reads a non-erased x and continues by entering an erased (maybe empty) segment on the right (performing an MV_R–instruction). The behaviour of this segment, stored in the control unit of M in B^R, determines the following possibilities (due to nondeterminism, more of them can occur at the same time):

a) E can cycle in the erased segment or halt in a non-accepting state; then M will have a possibility to reject,

b) E can halt within the erased segment in an accepting state; then M will have a possibility to accept,

c) E can leave the erased segment to the right in a state q; then M will be able to do a step which can be described as follows:

   – push the behaviour B^L and the symbol x into the pushdown store,
   – put (the value of) B^R in B^L and B_0 in B^R,
   – read the next input symbol and store it in $CSym$,
   – store q in CSt,

d) E can leave the erased segment to the left in a state q; then M will only store q in CSt.

Now consider the case when E performs an ER_L–instruction erasing a symbol between two erased segments with behaviours B^L, B^R; thus a longer (composed) segment arises. There are the following possibilities (it should be clear that they can be determined from B^L, B^R):

a) E can cycle or halt in the composed erased segment; then M will have possibilities to reject or to accept similarly as in the above (MV_R–) case,

b) E can leave the composed segment to the right in a state q; then M has a step which stores $Conc(Conc(B^L, B_1), B^R)$ in B^L, B_0 in B^R, reads the next input symbol storing it in $CSym$, and stores q in CSt,

c) E can leave the composed segment to the left in a state q; then M has a possibility to do two steps which store $Conc(Conc(B^L, B_1), B^R)$ in B^R, move the top pushdown symbol into $CSym$, move the next top pushdown symbol (which is a behaviour, maybe B_0) into B^L, and store q in CSt.

We leave out other technical details (e.g. the special case of the right sentinel \$, which can be handled by remembering in the control unit if \$ has been visited already). Nevertheless the given ideas should be convincing enough to show that a pushdown automaton M with $L(M) = L(E)$ can be really constructed. □

Now we proceed to show the inclusion $CFL \subseteq \mathcal{L}(MV_R, ER)$

We will show how to construct an (MV_R, ER)– automaton E recognizing a context-free language L given by a context-free grammar G. The construction will, in addition, ensure that E erases any item at the second visit at latest; we will come back to this fact in Section 5.

First we outline the main ideas informally.

For any input word w, E has to decide (in a nondeterministic way) whether $w \in L$ or not. It achieves this aim by performing a kind of the bottom-up analysis. Starting from the initial configuration with w on the list, E proceeds in phases (multisteps); in each phase, it chooses a grammar rule and tries to replace a substring corresponding to (a code of) the righthand side of the rule with a code of the nonterminal on the lefthand side of the rule. In this way, E aims to derive a code of the starting nonterminal S (thus proving $w \in L$).

Codes of nonterminals can only be constructed of (original) terminals in w and of (some occurrences of) symbol @. We represent nonterminals by natural numbers and code numbers in a natural way: number n will be coded by a terminal string of length n bounded by nonempty strings of symbols @; for the sake of unambiguity, each code starts with an additional terminal (e.g. $a_1@@a_2a_3a_4@@@$, a_i being terminals, is a code of number 3). In fact, a coded number will carry information not only about the relevant nonterminal but also about the terminal at the beginning of the code; the reason is technical – it helps to minimalize the number of visits at non-erased items.

Obviously, it must be guaranteed that the relevant code can always be created from the code of the righthand side of the chosen rule by erasing some terminals (i.e. by replacing them with @).

To guarantee this, it suffices that the righthand side of each grammar rule contains a sufficiently long terminal substring; it is clear that we can also admit a finite number of exceptions $S \to u$, u being a (short) terminal string, on condition that we exclude the starting nonterminal S from the righthand sides of all rules.

The mentioned sufficient condition can be ensured due to Lemma 3.3. which allows to suppose G in a $TS(k)$-form for sufficiently large k (it means that, with the above-mentioned exceptions $S \to u$, the righthand side of each rule of G finishes by at least k terminals – i.e. it has a Terminal Suffix of length k at least).

It should be clear from the above description how E recognizes L. If, in addition, E chooses righthand sides to be replaced in a systematic way (in the reversed order according to a right derivation of w in G when $w \in L$), it can be ensured that E always erases when moving left.

We proceed by formalizing the mentioned notions and ideas.

Definition 3.2. A context-free grammar $G = (V_N, V_T, S, R)$ is in a $TS(k)$ *-form* iff S does not occur on the righthand side of any rule in R and each rule in R is in one of the following two forms:

1/ $S \to u$, $u \in (V_T)^*$,

2/ $A \to \alpha u, A \in V_N, \alpha \in (V_N \cup V_T)^*, u \in (V_T)^*, length(u) \geq k.$

(In 1/ we can suppose $length(u) < k$ without changing the definition.)

Lemma 3.3. *For any context-free grammar $G = (V_N, V_T, S, R)$ and any natural number k , an equivalent context-free grammar $G' = (V'_N, V_T, S, R')$ in a $TS(k)$-form can be constructed; in addition, we can suppose the cardinality of V'_N to depend on G only, i.e. to be constant in variable k.*

Proof. By induction on k.

For $k = 1$, we can transform G into G' in a "reversed Greibach normal form" (each rule of the form $A \to \alpha a$, $\alpha \in (V'_N)^*$,$a \in V_T$) proceeding similarly as in the standard transformation into a Greibach normal form; in addition, we can exclude S from all righthand sides (using a new nonterminal).

In the induction step, we use the following two transformations which obviously keep the generated language and do not add new nonterminals:

1/ Let $B \to \alpha_1$, $B \to \alpha_2$,..., $B \to \alpha_m$ be all rules with B on the lefthand side. Then any rule $A \to \beta B \gamma$ can be replaced by the collection of rules $A \to \beta\alpha_1\gamma$, $A \to \beta\alpha_2\gamma$,...,$A \to \beta\alpha_m\gamma$.

2/ Any rule $A \to \alpha$ can be left out on condition that each other rule r is replaced by 2^n rules, n being the number of occurrences of A in the righthand side of r, where each of the new rules arises by replacing some (maybe none) occurrences of A with α in the righthand side of r.

Now suppose $G = (V_N, V_T, S, R)$ in a $TS(k)$-form, $k \geq 1$. It suffices to show an equivalent $G' = (V_N, V_T, S, R')$ (with the same set of nonterminals) in a $TS(k+1)$-form. The construction of G' is clear from the following two points:

– the rules of form $A \to \alpha B u, B \in V_N, u \in V_T^*, length(u) = k$, can be replaced as in 1/; the new rules obviously satisfy $TS(k+1)$-condition,

– the rules $A \to u, u \in V_T^*, length(u) = k$, can be left out as in 2/; any rule satisfying $TS(k+1)$-condition is replaced by rules, which satisfy $TS(k+1)$-condition as well.

□

Proposition 3.4. *Any context-free language is an (MV_R, ER)–language.*

Proof. Let a context free language L be generated by a context- free grammar $G = (V_N, V_T, S, R)$. Due to Lemma 3.3. we can suppose G in a $TS(k)$-form for $k = card(V_N \times V_T) + 3$.

First we define some technical notions.

For any natural number n, let any word of the regular language $V_T@^+(V_T)^n@^+$ be *a code of* n. It is easy to verify that if any $w \in (V_T \cup \{@\})^*$ can be interpreted as a sequence of terminals and codes of numbers, i.e. w is *decodable*, then it can be done unambiguously.

We only consider codes of numbers less or equal to $card(V_N \times V_T)$. Let these numbers be in one-to-one correspondence with the pairs (A, a), where A is from V_N, a is from V_T; by *a code of* (A, a) we mean a code of the corresponding number.

We say that a code of a number is *correct* if its first symbol is the terminal of the corresponding pair. A decodable w is *correct* if all codes in the sequence are correct.

For a decodable w, $DECOD(w)$ denotes the sequence which arises from w by replacing each code of a number with the nonterminal of the corresponding pair. A decodable w from $(V_T \ \cup \ \{@\})^*$ is *a code of* α *from* $(V_N \ \cup \ V_T)^*$ if $\alpha = DECOD(w)$.

Now we describe an (MV_R, ER)-automaton E recognizing L. At the beginning E

- either decides that the input word w is shorter than k and verifies the correspondence with some rule $S \to w$, w being terminal string,
- or proceeds as follows. E tries to simulate a right grammar derivation of the input word w in the reversed order. It proceeds in phases corresponding to derivation steps.

A general description of a phase is given in the following three points.

1/ E moves the head a guessed number times (maybe zero times) to the right.

2/ It guesses a grammar rule in a form $A \to \alpha u$, where α is either empty or ends by a nonterminal and u is from $(V_T)^*$, $length(u) \geq k$, and supposes that the head stands just before u (from the left).

If α is empty, E moves the head to the right as far as the last symbol of u (verifying that it really reads u); at the same time it replaces some (terminal) symbols by @ so as u is transformed into a code of (A, a), a being the first terminal of u.

If α is not empty, E moves the head to the left verifying that a correct code of α is read (from right to left); at the same time the head replaces every visited terminal with @. The only exception is the first (terminal) symbol of the code of α which is not entered (the head stops just before the symbol from the right) and therefore not verified. E remembers that, say, the terminal a should precede (a is the first symbol of α if α begins with a terminal; if α begins with a nonterminal B, a is determined by the relevant code of the pair (B, a)). Then E moves the head to the right and verifies that the substring of symbols @ is followed by u; by doing this, E leaves unchanged a substring of u (e.g. the beginning one) the length of which corresponds to the pair (A, a), and erases the rest of u.

3/ If $A = S$ in the rule guessed in 2/, E adds the verification that a correct code of (S, a) for some a has been left on the list, by which it finishes successfully.

As usual, if E meets a fault at any of the mentioned verifications, it immediately finishes with a failure.

It is easy to see that if w is from L (hence $S \Rightarrow^* w$ by G) then E accepts w. The opposite direction is clear from the following facts:

- the initial word w on the list is decodable and correct,

– if the word w' on the list before a phase as well as w'' after the phase are decodable and correct then $DECOD(w'') \Rightarrow_G DECOD(w')$,

– after each (successfully finished) phase, if the word contained on the list is not decodable and correct then the further computation must fail (any wrong guesses are later recognized).

Thus the proof is finished. Notice also that each terminal symbol is erased at the second visit of the head at latest. □

Propositions 3.1., 3.4 immediately imply

Theorem I. *$\mathcal{L}(MV_R, ER)$ is equal to CFL.*

4 $\mathcal{L}(MV_R, DL)$ is a proper subset of CFL

Since erasing can simulate deleting, Proposition 3.1. implies that any (MV_R, DL)–language is context-free. On the other hand, we will show that the context-free, even deterministic context-free, language

$$L = \Big\{ d^{n_1} a_{i_1} d^{n_1} d^{n_2} a_{i_2} d^{n_2} \ldots d^{n_k} a_{i_k} d^{n_k} c b_{i_k} b_{i_{k-1}} ... b_{i_1} \mid$$

$$k \geq 0,\ i_j \in \{1, 2\},\ n_j \geq 0 \text{ for } j = 1, 2, \ldots, k \Big\}$$

is not an (MV_R, DL)–language.

As any DL_R–instruction can be simulated by a DL_L– and two MV_R–instructions, it suffices to show that no (MV_R, DL_L)–automaton recognizes L.

To this aim, we characterize computations of (MV_R, DL_L)–automata by certain tree structures, called computation trees, and use two pumping lemmas for them.

Let D be an (MV_R, DL_L)–automaton. Without loss of generality, we assume that D visits all items of the input list during any (complete) computation. A computation of D can be represented by a labelled directed graph, called a *computation graph*: the nodes are items of the input list labelled by the corresponding input symbols; any moving head from item u_1 to item u_2 according to an instruction I will correspond to an edge labelled by I and leading from the node u_1 to the node u_2.

In our case, the structure of a computation graph is very lucid. If we omit the edges corresponding to DL_L–instructions, we get a (directed) tree, and the whole graph corresponds, in fact, to the left-to-right traversal of the tree. Note that, during the traversal, the items are visited in the order corresponding to the natural (left-to-right) order in the input list. The only technical complication is connected with the right sentinel \$, where MV_L–instructions are applicable. But it is clear that we can suppose that D visits \$ at most once (afterwards it is remembered in the control unit) and we can handle these MV_L–instructions as the DL_L–ones.

The mentioned ideas lead to the following definition.

Definition 4.1. Let D be an (MV_R, DL_L)–automaton and C a computation of D on an input list $¢\, a_1 a_2 ... a_n \$$. The *computation tree* $T(C)$ corresponding to C is the following rooted (directed) tree:
– The nodes of $T(C)$ are the items of the input list labelled successively by $¢, a_1, ..., a_n, \$$. The natural "left-to-right" ordering of items, denoted by $<$, can thus be used for the nodes of $T(C)$, too.
– $T(C)$ contains an edge (u_1, u_2) iff D moved in C from the item u_1 to the item u_2 according to an MV_R–instruction (obviously $u_1 < u_2$). This instruction is a part of the *history* of (u_1, u_2); the other part of the history is the DL_L–instruction according to which D moves from u_2 to u_1, or is empty if it is not the case.

A subtree T_1 of $T(C)$ is a *complete subtree* if any node u between two nodes u_1,u_2 of T_1 ($u_1 < u < u_2$) is also in T_1.

By $CT(D)$ we will denote the set of all computation trees corresponding to the accepting computations of D.

Remark. Notice that traversing a complete subtree, starting and finishing at the root, means making a "gap" in the input list of length equal to the number of non-root nodes in the subtree.

Now we show two pumping lemmas for the computation trees, which should be clear from the illustrating figures.

Lemma 4.2.(First pumping lemma; cf. Fig.1 and Fig.2)
Let T be from $CT(D)$ for some (MV_R, DL_L)–automaton D.
Let (u_1, u_1'), (u_2, u_2') be edges of T with the same history and let u_1' and u_2 be connected by a (directed) path. Let T_1 be the maximal subtree of T with root u_1' and T_2 the maximal subtree of T with root u_2'.
Then:
a) T' created from T by replacing T_2 with T_1 is from $CT(D)$ (the corresponding input word was "pumped").
b) (Cutting variant) T'' created from T by replacing T_1 by T_2 is from $CT(D)$ (the corresponding input word was "cut").

Figure 1

Figure 2

Lemma 4.3.(Second pumping lemma; cf. Fig.3 and Fig.4)
Let T be from $CT(D)$ for some (MV_R, DL_L)–automaton D. Let (u_1, u_2), (u_1, u_3), where $u_2 < u_3$ be edges of T with the same history. Let T_{23}^1 be a subtree of T, consisting of nodes u_1, u_2 and all nodes u such that $u_2 < u < u_3$.
Let T' be created from T by inserting a new copy of T_{23}^1 between the old copy and the edge (u_1, u_3) identifying the roots of both copies (u_1 will be the common root). Then T' is from $CT(D)$ (the corresponding input word was "pumped").

Figure 3

Figure 4

Now we are ready to show the desirable result for L defined at the beginning of the section.

Proposition 4.4. *L is not recognized by any (MV_R, DL)–automaton.*

Proof. We show it by contradiction. Suppose there is an MV_R, DL–automaton D recognizing L. As mentioned above, we can suppose that D does not have DL_R–instructions. Let i denote the number of its instructions; note that the number of possible histories of edges is bounded by $h = i + i^2$.

Let T be the computation tree for some accepting computation of D on some input word $w \in L$.

Let T_d be any complete subtree of T with all non-root nodes labelled by d (T_d corresponds to a gap made in a d-segment of w).

Suppose there is a path in T_d which is longer than h or there is a node with more than h immediate successors. Then either the first or the second pumping lemma can be applied; the respective d–segment can thus be pumped so that we obtain a word from $L(D)$ which is not in L. Hence it is clear that the number of nodes in T_d is bounded by a constant k_d (depending on D only).

Suppose now that $w = ua_Ld^ma_Rv$, where a_L,a_R are from $\{a_1, a_2, c\}$ (a_L,a_R are some neighbour a–symbols or a_L is the rightmost a–symbol and $a_R = c$), and that there is a path from a_L to a_R in T. If $m > k_dh$ then the path must be longer than h due to the choice of k_d. Hence the cutting variant of the first pumping lemma can be used, possibly several times, to show that D accepts a word $ua_Ld^na_Rv'$ for $n \leq k_dh$.

Therefore when we put $r = k_dh + 1$ and define L_p (for any $p \geq 1$) as the set of all words from L of the form

$$w = d^r a_{i_1} d^{2r} a_{i_2} \dots d^{2r} a_{i_p} d^r c b_{i_p} \dots b_{i_2} b_{i_1}$$

we can conclude that, for any $w \in L_p$, the corresponding tree T does not contain a path from a_L to a_R. The above mentioned cutting would lead to a word outside L (the word would have the form $ua_Ld^na_Rv$ where ua_L is a prefix of w and $n < r$).

It means that when, working on a $w \in L_p$, D visits the symbol c for the first time, all a–symbols have been deleted and the respective configuration, we call it *c–configuration*, can be described as $(\cent\, d^n, q, cu\$)$ where $n \leq 2rp$ ($u \in \{b_1, b_2\}^*$; q is the current state of D). There are 2^p words in L_p, and p could have been chosen so that $2^p > (2rp + 1)s$ where s is the number of states of D. Hence there surely are two different w_1, w_2 in L_p whose c–configurations in some accepting computations are $(\cent\, d^n, q, cu_1\$)$, $(\cent\, d^n, q, cu_2\$)$ respectively (note that $w_1 \neq w_2$ implies $u_1 \neq u_2$). Hence also the word arising from w_1 by replacing the suffix u_1 with u_2 is accepted by D; but this word is obviously not in L, which is the final contradiction. □

Hence we have proved Theorem II:

Theorem II. *$\mathcal{L}(MV_R, DL)$ is a proper subset of CFL.*

5 Additional Remarks

Inspired by [4] and [2], special complexity measures for forgetting automata can be considered. In particular we can define that an erasing (MV, ER)–automaton E has EH–complexity k if for any word $w \in L(E)$ there is an accepting computation during which each item is erased at the k-th visit of the head at latest. It follows from [4] that any language recognized by an erasing automaton with EH–complexity k (for any k) is context free. Hence the proof of Proposition 3.4. shows that context-free languages can be characterized by EH–complexity 2 (or any $k \geq 2$).

We mentioned that the power of deleting automata is more unclear. Our conjecture is that e.g. the language generated by the following context-free grammar is not recognized by any deleting (MV, DL)– automaton.

$S' \rightarrow ST$
$S \rightarrow aSA \mid bSB \mid a \mid b$
$T \rightarrow ATa \mid BTb \mid a \mid b$
$A \rightarrow aAa \mid bAb \mid a$
$B \rightarrow aBa \mid bBb \mid b$

The proof (if the conjecture is correct) would probably need more sophisticated methods than those of Section 4; since (general) MV_L–instructions are allowed, the computation graphs are no longer trees.

[7] shows further relations among classes of languages according to the instructions allowed. E.g. CFL is a proper subclass of $\mathcal{L}(MV_R, ER, DL)$; it is shown by using the language $\{a^n b^n c^n \mid n \geq 1\}$. $\mathcal{L}(MV_R, ER, DL)$ is a proper subclass of $\mathcal{L}(MV, ER, DL)$; this is shown by using the language $\{wcw \mid w \in \{a, b\}^*\}$.

A referee of this paper, as well as F.J.Brandenburg in a discussion with us, draw our attention to the next two remarks.

In [3] Greibach shows a context-free language L_G such that every context-free language is an inverse homomorphic image of L_G. Since $\mathcal{L}(MV_R, DL)$ is a proper subset of CFL and since the hardest context-free language L_G is obviously in $\mathcal{L}(MV_R, DL)$, the class $\mathcal{L}(MV_R, DL)$ is not closed under inverse homomorphism, and its closure under inverse homomorphism is CFL.

The second remark aims to another proof of the inclusion $CFL \subseteq \mathcal{L}(MV_R, ER)$. The language L_G is clearly in $\mathcal{L}(MV_R, ER)$, it only remains to prove that $\mathcal{L}(MV_R, ER)$ is closed under inverse homomorphism. But we do not have a simple evidence for it.

At the end we can mention that similar problems which were handled in the paper can be considered for deterministic automata as well. Let us only remark that it is open whether the class of languages recognized by deterministic (MV_R, ER)–automata is a proper subclass of $DetCFL$.

References

1. von Braunmühl B., Verbeek R.: Finite change automata; Proceedings of the Fourth GI Conference on Theoretical Computer Science, Lect. Notes in Comp. Sci., Vol.67, Springer-Verlag, Berlin, 1979, pp.91–100
2. Chytil M.P., Karlgren H.: Categorial grammars and list automata for strata of non-CF-languages; in Buszkowski W., Marciszewski W., van Benthem J. (eds.): Categorial Grammars, Linguistic and literary studies in Eastern Europe 25, 1988
3. Greibach S.A.: The hardest context-free language, SIAM Journal on Computing 2, 1973, pp. 304–310
4. Hibbard T.N.: A generalization of context-free determinism, Information and Control 11, 1967, pp.196–238
5. Jančar P.: Nondeterministic Forgetting Automata are Less Powerful than Deterministic Linear Bounded Automata; Acta Math. et Inf. Univ. Ostraviensis, 1, Ostrava, 1993, pp.67–73
6. Jančar P., Mráz F., Plátek M.: Characterization of Context-Free Languages by Erasing Automata; in Proc. MFCS'92, Prague, Czechoslovakia, August 1992, Lect. Notes in Comp. Sci., Vol. 629, Springer, Berlin 1992, pp.305–314
7. Jančar P., Mráz F., Plátek M.: A taxonomy of forgetting automata; in Proc. MFCS 1993, Gdańsk, Poland, September 1993, Lect. Notes in Comp. Sci., Vol. 711, Springer 1993, pp.527–536
8. Marcus S.: Algebraic Linguistics; Analytical Models. New York and London 1967
9. Plátek M., Vogel J.: Deterministic List Automata and Erasing Graphs; The Prague bulletin of mathematical linguistics 45, 1986