Collapse Operation Increases Expressive Power of Deterministic Higher Order Pushdown Automata

Pawel Parys

▶ **To cite this version:**

Pawel Parys. Collapse Operation Increases Expressive Power of Deterministic Higher Order Pushdown Automata. Symposium on Theoretical Aspects of Computer Science (STACS2011), Mar 2011, Dortmund, Germany. pp.603-614. hal-00573593

HAL Id: hal-00573593
https://hal.science/hal-00573593

Submitted on 4 Mar 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Collapse Operation Increases Expressive Power of Deterministic Higher Order Pushdown Automata*

Paweł Parys

University of Warsaw
ul. Banacha 2, 02-097 Warszawa,Poland
parys@mimuw.edu.pl

Abstract

We show that collapsible deterministic second level pushdown automata can recognize more languages than deterministic second level pushdown automata (without collapse). This implies that there exists a tree generated by a second level recursion scheme which is not generated by any second level safe recursion scheme.

1998 ACM Subject Classification F.4.3 Formal Languages

Digital Object Identifier 10.4230/LIPIcs.STACS.2011.603

1 Introduction

In verification we often approximate an arbitrary program by a program with variables from a finite domain, remembering only a part of information. Then the outcome of some conditions in the program (e.g. in the *if* or *while* statements) cannot be determined, hence they are replaced by a nondeterministic choice (branching). If the program does not use recursion, the set of its possible control flows is a regular language, and the program itself is (in a sense) a deterministic finite automaton recognizing it. If the program contains recursion, we get a deterministic context free language, and from the program one can construct a deterministic pushdown automaton (PDA for short) recognizing this language. In other words, stack can be used to simulate recursion (notice that the same is true for compilers: they convert a recursive program into a program using stack). In verification it is interesting to analyze the possibly infinite tree of all possible control flows of a program. This tree has a decidable MSO theory [4].

A next step is to consider higher order programs, i.e. programs in which procedures can take procedures as parameters. Such programs closely correspond to so-called higher order recursion schemes and to typed λ-terms. They no longer can be simulated by classical PDA. Here higher order PDA come into play. They were originally introduced by Maslov [10]. In automata of level n we have a level n stack of level $n-1$ stacks of ... of level 1 stacks. The idea is that the PDA operates only on the topmost level 1 stack, but additionally it can make a copy of the topmost stack of some level, or can remove the topmost stack of some level. However the correspondence between higher order automata and recursion schemes (programs) is not perfect. Trees recognized (in suitable sense) by a deterministic PDA of level n coincide with higher order recursion schemes of level n with *safety* restriction [7]. See [3, 5] for another characterizations of the same hierarchy. It is important that these trees have decidable MSO theory [7].

To overcome the safety restriction, a new model of pushdown automata were introduced, called collapsible higher order PDA [8, 1]. These automata are allowed to perform an

* Work supported by Polish government grant no. N206 008 32/0810.

© Paweł Parys;
licensed under Creative Commons License NC-ND
28th Symposium on Theoretical Aspects of Computer Science (STACS'11).
Editors: Thomas Schwentick, Christoph Dürr; pp. 603–614

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

additional operation called *collapse* (or *panic* in [8]); it allows to remove all stacks on which a copy of the currently topmost stack symbol is present. These automata correspond to all higher order recursion schemes (not only safe ones) [6], and trees generated by them also have decidable MSO theory [11]. It is also worth to mention that verification of some real life higher order programs can be performed in reasonable time [9].

A question arises if these two hierarchies are possibly the same hierarchy? This is an open problem stated in [7] and repeated in other papers concerning higher order PDA [8, 2, 11, 6]. We give a negative answer to this question, which is our main theorem.

▶ **Theorem 1.** *There exists a language recognized by a collapsible deterministic second level pushdown automaton, which is not recognized by any deterministic second level pushdown automaton without collapse.*

From the equivalences mentioned above we get the following.

▶ **Corollary 2.** *There exists a tree generated by a second level recursion scheme, which is not generated by any safe second level recursion scheme.*

This confirms that the correspondence between higher order recursion schemes and deterministic higher order PDAs is not perfect. The language used in Theorem 1 comes from [7] and from that time was conjectured to be a good candidate.

Related work

One may ask a similar question for nondeterministic automata rather than for deterministic ones. This is an independent problem. The answer is known only for level 2 and is opposite. One can see that for level 2 the collapse operation can be simulated by nondeterminism, hence normal and collapsible nondeterministic level 2 PDA recognize the same languages [2]. However it seems that in context of verification considering deterministic automata is a more natural choice, for the following reasons. First, most problems for nondeterministic PDA are not decidable: even the very basic problem of universality for level 1 PDA is undecidable. Second, we want to verify deterministic programs (possibly with some not deterministic input). A nondeterministic program is something rather strange: it has an oracle which says what to do in order to accept. Normally, when a program is going to make some not deterministic choice, we want to analyze all possibilities, not only these which are leading to some „acceptance" (hence we have branching, not nondeterminism).

2 Definition

A *deterministic second level pushdown automaton* (D2PDA for short) is given by a tuple $(A, \Gamma, \gamma_I, Q, q_I, \delta)$ where A is an input alphabet, Γ is a stack alphabet, γ_I is an initial stack symbol, Q is a set of states, q_I is an initial state, and $\delta\colon Q \times \Gamma \to Ops$ is a transition function. The set Ops contains the following operations: (pop, q), $(\mathsf{push}(\gamma), q)$, (copy, q), $\mathsf{read}_0(t)$, $\mathsf{read}_{acc}(t)$ for each $q \in Q$, $\gamma \in \Gamma$, and $t\colon A \to Q$.

A first level stack is a nonempty sequence of elements of Γ. A second level stack is a nonempty sequence of first level stacks. A configuration of a D2PDA consists of a second level stack, a state from Q, and a head position over the input word. At the beginning on the second level stack there is one first level stack, which contains one γ_I symbol, the state is q_I, and the head is before the first letter of the input word. The automaton always sees only the last (topmost) symbol on the last (topmost) stack. When the current state is q and the last stack symbol is γ, the automaton looks at the transition $\delta(q, \gamma)$ and:

- if $\delta(q,\gamma) = (\mathsf{pop}, q')$, it removes the last symbol from the last first level stack; when the stack becomes empty, it is removed from the second level stack; when the second level stack becomes empty, the automaton fails; the state becomes q';
- if $\delta(q,\gamma) = (\mathsf{push}(\gamma'), q')$, it places symbol γ' on the end of the last first level stack; the state becomes q';
- if $\delta(q,\gamma) = (\mathsf{copy}, q')$, it places a copy of the last first level stack on the end of the second level stack; the state becomes q';
- if $\delta(q,\gamma) = \mathsf{read}_0(t)$, it moves the head to the next letter of the input word; if it is a, the state becomes $t(a)$; if we are on the end of the word, the automaton fails;
- if $\delta(q,\gamma) = \mathsf{read}_{acc}(t)$, it moves the head to the next letter of the input word; if it is a, the state becomes $t(a)$; if we are on the end of the word, the automaton accepts the word.

Notice that none of the stacks is empty; if the last element of a stack is removed, we remove also the whole stack from the second level stack.

Now we are going to define *collapsible D2PDA*. Its first level stacks together with each symbol $\gamma \in \Gamma$ contain a number $n \in \mathbb{N}$ (hence stacks contain pairs (γ, n)). The operation $\mathsf{push}(\gamma)$ places a pair (γ, n) on the end of the last first level stack, where n is the number of first level stacks. We additionally have an operation $(\mathsf{collapse}, q')$ for each $q' \in Q$. When the last element of the last stack is (γ, n) and this operation is performed, we remove all stacks except the first $n-1$ stacks; if $n-1=0$ the automaton fails; the state becomes q'. In other words, $\mathsf{collapse}$ removes all stacks on which a copy of this (γ, n) symbol is present.

An example of a collapsible D2PDA is given in the next section. In the literature one can find some slightly different definitions of a D2PDA and a collapsible D2PDA, but one can see that they are equivalent to ours, through some encodings.

3 The language

Let $A = \{[,], *\}$. A word $w \in \{[,]\}^*$ is called a *prefix of a bracket expression* if in each prefix v of w the number of closing brackets is not greater than the number of opening brackets. A word $w \in \{[,]\}^*$ is called a *bracket expression* if it is a prefix of a bracket expression and the number of opening brackets in w is equal to the number of closing brackets in w. Let PBE and BE be the set of all prefixes of bracket expressions and of all bracket expressions, respectively. For $w \in PBE$ by $open(w)$ we denote the number of [characters in w minus the number of] characters in w (i.e. the number of opened brackets which are not closed). For each $w \in PBE$ we define a number $char(w)$ as $|w| - |v|$ where v is the longest suffix of w, which is a bracket expression. This number is called later a *characteristic* of the word w. We consider the following language over A:

$$U = \{w*^{char(w)+1} \colon w \in PBE\}.$$

The words $[][[]****$, $[[[****$, $[]*$ are examples of words in U, and $[[[***$ and $[]]*$ are examples of words not in U (moreover, no word beginning with $[[]$ is in U).

It is known that U can be recognized by a collapsible D2PDA, but for completeness we show it below. The collapsible D2PDA will use three stack symbols: X (used to mark the bottom of stacks), Y (used to count brackets), Z (used to mark the first stack). Initially, the only stack contains one X symbol. The automaton first pushes Z, makes a copy, and pops Z (hence the first stack is marked with Z, the other stacks are used later). Then, for an opening bracket we push Y and we make a copy; for a closing bracket we pop Y and we make a copy. Hence for each bracket we have a stack and on the last stack we have as many

Y symbols as the number of currently open brackets. If for a closing bracket the topmost symbol is X, we fail: it means that the input is not a prefix of a bracket expression.

Finally a star is read. If the topmost symbol is X, we have read a bracket expression, hence we should accept after one star. Otherwise, the topmost Y symbol corresponds to the last opening bracket which is not closed. We do the collapse operation. It leaves the stacks corresponding to the earlier brackets (and the first stack), hence the number of stacks is precisely equal to the characteristic. Now we should read as many stars as we have stacks, and accept (after each star we remove one stack).

4 The proof

In this section we show that U is not recognized by any D2PDA. Assume otherwise: there exists a D2PDA $\mathcal{A}$ recognizing U.

By $\sim$ we denote the Myhill-Nerode relation with respect to U: we have $v \sim w$ if for all u it holds $vu \in U \Leftrightarrow wu \in U$. Notice, in particular, that $open(v) \neq open(w)$ implies $v \not\sim w$.

Our first goal is to eliminate situations in which the number of stacks decreases. As a first step we will eliminate situations in which the number of stacks is first increased and later after a long time decreased to the same value. The intuition is as follows. Consider a run of $\mathcal{A}$ on a word w and a moment when the number of stacks increases from some s to $s+1$. It happens when the head is over some position i of the input word. Let j be the position of the head in the moment when the number of stacks becomes again s (for the first time). Assume that such j exists and that $j-i$ is big. This can happen only if we have a very bad luck. Indeed, consider a word w' get from w by some modification between positions i and j, and assume that in w' the number of stacks also comes down to s at some moment. Notice that in the moment when the number of stacks becomes s, we have the same stacks content for w and for w', the only difference is the state. We have only a fixed number of states and very many nonequivalent modifications of w, which have to give a different state. Hence in most cases either the number of stacks goes down to s very quickly after i, or it stays always above s.

It is formalized using fillings. We say that a function $\sigma\colon PBE \to PBE$ is a *filling*, when

- $\sigma(\epsilon) = \epsilon$, and
- for any $vb \in PBE$ (where $b \in \{[,]\}$), it holds that $\sigma(vb) = \sigma(v)eb$ for some bracket expression e (which may depend on both v and b).

Hence a filling of a word is received by inserting a bracket expression before each letter, but in a deterministic way. A filling is called a *k-filling*, when additionally the length of each inserted bracket expressions e is at most k. We have the following lemma.

▶ **Lemma 3.** *Assume $\mathcal{A}$ is a D2PDA recognizing U. Then there exist constants k, l and a k-filling σ such that if $\mathcal{A}$ reading $\sigma(w)$ for some $w \in PBE$ increases the number of stacks from some s to $s+1$ with the head over a position i, then either the number of stacks is decreased to s for the head over a position $\leq i+l$, or it stays above s for the rest of the run.*

Proof. The constant $k = O(|Q|^2)$ will follow from the proof; we take $l = 2(k+1)$. Our filling σ will satisfy the following additional assumption for any $w \in PBE$:

- ⋆ Let s_0 be the minimal number of stacks when the head of $\mathcal{A}$ is over one of the last $k+1$ positions of $\sigma(w)$. Then for any $e \in PBE$, the number of stacks never goes below s_0 while $\mathcal{A}$ reads the suffix $[e$ of $\sigma(w)[e$.

We will not define the filling explicitly. Instead, we define it in a non explicit way by induction. We construct the values of filling σ starting from shorter words and going towards longer.

For the empty word we take $\sigma(\epsilon) = \epsilon$; property $\star$ is satisfied ($s_0 = 1$ and we can not have less than one stack), as well as the thesis of the lemma.

Now consider any longer word $vb \in PBE$ (where b is a letter). Let first define some words. For any number n and any function $f\colon \{1,\dots,n\} \to \{0,1\}$ we define inductively

$$o_f^n = \begin{cases} \epsilon & \text{when } n = 0, \\ o_f^{n-1}[& \text{when } f(n) = 0, \\ o_f^{n-1}[][& \text{when } f(n) = 1. \end{cases}$$

Hence o_f^n consists of n opening brackets, and before i-th of them we insert $[\,]$ if $f(i) = 1$. Let s_0 be the minimal number of stacks when the head of $\mathcal{A}$ is over one of the last $k+1$ positions of $\sigma(v)$. Let s_1 be the number of stacks after $\sigma(v)$ is read (precisely, in the moment of the read operation moving the head from the last letter of $\sigma(v)$). Let d be the greatest number ($s_0 \le d \le s_1$) such that for some $f\colon \{1,\dots,|Q|+1\} \to \{0,1\}$ the number of stacks never goes below d while $\mathcal{A}$ reads the added suffix of $\sigma(v)[o_f^{|Q|+1}$ (by the *added suffix* we mean the part after $\sigma(v)$). From the $\star$ property it follows that s_0 satisfies this, hence d exists. Fix the particular function f, for which the number of stacks never goes below d while $\mathcal{A}$ reads the added suffix of $\sigma(v)[o_f^{|Q|+1}$.

Consider the functions $f_1,\dots,f_{|Q|+1}$ which differ from f only on one position, namely $f_i(i) \neq f(i)$ and $f_i(j) = f(j)$ for all $j \neq i$. We will show that for at most $|Q|$ of them the number of stacks goes below d while $\mathcal{A}$ reads the added suffix of $\sigma(v)[o_{f_i}^{|Q|+1}e$ for some $e \in PBE$. To see this, for each such f_i fix some e_i (if exists) such that the number of stacks goes below d while $\mathcal{A}$ reads the added suffix of $\sigma(v)[o_{f_i}^{|Q|+1}e_i$. Consider any two such functions f_i and f_j ($i < j$). Let x_i be the prefix of $\sigma(v)[o_{f_i}^{|Q|+1}e_i$ such that the number of stacks decreases to $d-1$ when the head is over the last letter of x_i. Similarly for j. The key point is that neither x_i nor x_j can be a prefix of $\sigma(v)[o_f^{|Q|+1}$ (i.e. x_i has to contain some letters which are different for f_i than for f), as for this word the number of stacks stays at least d. Assume first that $x_i = \sigma(v)[o_{f_i}^{i-1}[\,]$ (which is possible for $f_i(i) = 1$). But x_j contains at least $\sigma(v)[o_{f_j}^{j-1}$, so $\mathit{open}(x_i) < \mathit{open}(x_j)$, hence $x_i \not\sim x_j$ (recall that $\sim$ is the Myhill-Nerode relation). The other case is that x_i contains at least $\sigma(v)[o_{f_i}^{i}$. When $\mathit{open}(x_i) \neq \mathit{open}(x_j)$, we also have $x_i \not\sim x_j$. When $\mathit{open}(x_i) = \mathit{open}(x_j)$, consider z which closes $\mathit{open}(x_i) - \mathit{open}(\sigma(v)[o_{f_i}^{i}$ brackets. We have $\mathit{char}(x_iz) = |\sigma(v)[o_{f_i}^{i}|$ and $\mathit{char}(x_jz) = |\sigma(v)[o_{f_j}^{i}| = \mathit{char}(x_iz) \pm 2$, hence in this case also $x_i \not\sim x_j$. This means that in the moment when the number of stacks becomes $d-1$, the state has to be different for i and j (as the stacks content is the same, but the read inputs are not equivalent). As we have only $|Q|$ states, the number of stacks may become $d-1$ only for at most $|Q|$ functions f_i. Thus there is g (one of $f_1,\dots,f_{|Q|+1}$) such that for each $e \in PBE$, the number of stacks stays at least d while $\mathcal{A}$ reads the added suffix of $\sigma(v)[o_g^{|Q|+1}e$.

Now consider the words (in BE)

$$u_i = \sigma(v)[o_g^{|Q|+1}[^i]^{i+|Q|+2}b$$

for i being a multiple of $|Q|+3$. We will show that for at most $|Q|$ of them the number of stacks goes below d while $\mathcal{A}$ reads their added suffixes. To see this take any two such words u_i and u_j ($i < j$). Let x_i be the prefix of u_i such that the number of stacks decreases to $d-1$ when the head is over the last position of x_i; similarly x_j for u_j. We know from the above that the number of stacks can not be decreased to $d-1$ inside $[o_g^{|Q|+1}[^j]^j$ (it is true for $[o_g^{|Q|+1}e$ for any $e \in PBE$, in particular for $e = [^j]^j$), hence $|x_j| \ge |u_j| - |Q| - 2$. However $|u_j| \ge |u_i| + |Q| + 3$ and $|u_i| \ge |x_i|$, which gives $|x_j| > |x_i|$. Thus the characteristics of $x_i[$

and $x_j[$ are different, $x_i \not\sim x_j$. This means that the number of stacks is decreased to $d-1$ in a different state in these two words. Thus only in $|Q|$ words u_i the number of stacks may go below d.

Observe also that there are at most $|Q|$ words u_i in which for some $e \in PBE$ the number of stacks goes below d in the part $[e$ of $u_i[e$, but stays at least d inside the added suffix of u_i. To see this, for each such i fix some $e_i \in PBE$ (if exists) such that the number of stacks goes below d in the part $[e_i$ of $u_i[e_i$. We may assume that this happens when the head is over the last letter of $u_i[e_i$; otherwise the last letter of e_i is redundant and can be cut off. Take any two such words u_i and u_j $(i < j)$. If $open(e_i) \neq open(e_j)$, we have $u_i[e_i \not\sim u_j[e_j$. Otherwise, let z consist of $open(e_i)$ closing brackets; see that $char(u_i[e_i z) = |u_i[|$ and $char(u_j[e_j z) = |u_j[|$. But the lengths of u_i and u_j are different, hence $u_i[e_i \not\sim u_j[e_j$. Thus when the number of stacks is decreased to $d-1$, the state for i and for j has to be different. As we have only $|Q|$ states, there are at most $|Q|$ such words.

From the above two paragraphs it follows that we may choose u_i for $i \leq (2|Q|+1)(|Q|+3)$ such that for each $e \in PBE$ the number of stacks stays at least d while $\mathcal{A}$ reads the added suffix of $u_i[e$ (both inside and outside u_i). As k we take the maximal length of the expression inserted for any such u_i. Observe that this u_i satisfies both the thesis of the lemma and property $\star$. Indeed, whenever the number of stacks decreases from some $s+1$ to s during the added suffix of u_i, then $s \geq d \geq s_0$, hence the number of stacks was increased from s to $s+1$ during the last $l = 2(k+1)$ letters of u_i (and when the decrease is inside $\sigma(v)$, everything is OK from the induction assumption). From the method how d was chosen follows that at some moment while $\mathcal{A}$ reads the added suffix of u_i, the number of stacks is d (even inside the $[o_g^{|Q|+1}$ fragment). On the other hand, for each $e \in PBE$ the number of stacks never goes below d while reading the part $[e$ of $u_i[e$. Thus the $\star$ property is also satisfied. ◀

The next lemma eliminates also all other situations in which the number of stacks is increased from some s to $s+1$ for the head over one position of the word, and then it is decreased from $s+1$ to s over any of the next positions of the word (not only farther than l letters).

▶ **Lemma 4.** *Assume there exists a D2PDA recognizing U. Then there exists a constant k, a k-filling σ and a D2PDA $\mathcal{A}'$ recognizing U such that if $\mathcal{A}'$ reading $\sigma(w)$ for some $w \in PBE$ increases the number of stacks from some s to $s+1$ with the head over a position i, then either the number of stacks is decreased to s for the head over the position i, or it stays above s for the rest of the run.*

Proof. Let $\mathcal{A}$ be a D2PDA recognizing U. The constant k and the k-filling σ are taken from Lemma 3. We have to improve $\mathcal{A}$ such that the stronger property will be satisfied. The automaton $\mathcal{A}'$ remembers a state q of $\mathcal{A}$ and up to l previous letters of the input (where l is the constant from Lemma 3), i.e. a state of $\mathcal{A}'$ contains a state of $\mathcal{A}$ and a sequence of up to l letters (called a *buffer*). We begin with the initial state of $\mathcal{A}$, and no letters in the buffer. When the number of remembered letters is smaller than l, we read the next letter and we append it to our buffer. When the buffer is full (contains l letters), we start executing $\mathcal{A}$. First, we execute $\mathcal{A}$ from the remembered state q until the moment when it reads a letter (we give him the first letter from the buffer). Then, consider also the further run of $\mathcal{A}$, which reads all the next letters of the buffer (until the moment when $\mathcal{A}$ wants to make a read operation when no more letters are in the buffer). We execute the part of this run up to the moment when the number of stacks is minimal (to the last such moment if there are more than one); we describe below how to detect this moment. Denote this minimal number

of stacks as s_0. Of course after a letter is read by the run of $\mathcal{A}$ being simulated, we remove it from the buffer.

When $\mathcal{A}'$ sees a star, it executes $\mathcal{A}$ on the letters left in the buffer, and then it simply emulates $\mathcal{A}$ on the rest of the word. Of course $\mathcal{A}$ and $\mathcal{A}'$ accept the same language, as they in fact perform the same operations on a given input word, the only difference is when the read operations are done (in $\mathcal{A}'$ we earlier read more letters and later perform the other operations). Note, in particular, that in the part reading brackets, $\mathcal{A}'$ may always use the read_0 operation, as all words in U have at least one star.

Observe that when such $\mathcal{A}'$ reads a word $\sigma(w)$, the thesis of our lemma is satisfied. Indeed, when the number of stacks is increased from some $s \geq s_0$ to $s+1$, then it decreases back to s before the head is moved (as the head is moved with s_0 stacks). On the other hand, when the number of stacks is increased from some $s < s_0$ to $s+1$, it is done by $\mathcal{A}$ before reading the first letter of the buffer. Later $\mathcal{A}$ does not decrease the number of stacks below s_0 (hence to s) when the head is over any of the next l positions. Thus $\mathcal{A}$ never does this (from the thesis of Lemma 3), hence $\mathcal{A}'$ also.

How to create such $\mathcal{A}'$? The difficulty is that $\mathcal{A}'$ has to find the moment in which the number of stacks is minimal. However it can be done. The part always executed (i.e. up to the first read) is executed in a normal way. Then the rest is executed, but each new stack (created by the copy operation) is marked by the state of $\mathcal{A}$ before the copy operation and by the head position of $\mathcal{A}$ (i.e. how many letters of the buffer were read). More precisely, for the last stack the marking is remembered in the state of $\mathcal{A}'$; for the previous stacks a special stack symbol is put on the top of a stack when the number of stacks is increasing, and is taken from the top of a stack after the number of stacks decreases. Finally, after the whole run reading the buffer is executed, we remove all the stacks with the markings. This gives us s_0 stacks (the minimal number of stacks during the second part of the run). The marking of the last removed stack gives us the new state q of $\mathcal{A}$, and the number of letters which should be removed from the buffer. ◀

In the next lemma we go even further and we eliminate all situations in which the number of stacks is decreased.

▶ **Lemma 5.** *Assume there exists a D2PDA recognizing U. Then there exists a constant k, a k-filling σ and a D2PDA $\mathcal{A}$ recognizing U such that $\mathcal{A}$ reading $\sigma(w)$ for some $w \in PBE$ never decreases the number of stacks.*

Proof. The constant k and the filling σ is taken from Lemma 4 (hence also from Lemma 3). Let $\mathcal{A}'$ be the automaton from Lemma 4; we will improve it, getting an automaton $\mathcal{A}$. We enrich the stack alphabet: together with each stack symbol we keep a function $f\colon Q \to Q \cup \{\mathsf{nr}\}$, where Q is the set of states of $\mathcal{A}'$. The function lying on an i-th place of an s-th stack is defined in the following way. Consider the situation when all stacks after s are removed and all symbols from the s-th stack above the i-th symbol are removed (i.e. the function lies on the topmost place of the last stack). Let start the automaton from a state q. We look at the run until it tries to do a read operation, or until the number of stacks is decreased to $s-1$. When the read operation is first, we assign $f(q) = \mathsf{nr}$. When the decrease is first, and it results in a state p, we assign $f(q) = p$. It is also possible that the run is infinite (it loops in some stupid way), then we also assign $f(q) = \mathsf{nr}$.

The claim is that we can modify the automaton $\mathcal{A}'$ (getting $\mathcal{A}''$) so that it puts on the stack the correct f function together with each symbol. This is because f lying together with some symbol somewhere on a stack depends only on this symbol and on the function

f one place below. In particular it depends only on the contents of the current stack, hence after making a copy of a stack, the functions in the copy stay correct.

Now we make one more modification of $\mathcal{A}''$, getting $\mathcal{A}$. Whenever $\mathcal{A}''$ is going to do the copy operation, we look at the f function of the topmost stack symbol. When $f(q) = \mathsf{nr}$ we really do the copy operation. Otherwise we immediately move to state $f(q)$ without any operation (formally, as each transition has to do some operation, we may for example push something to the stack and then pop it). We may do this, since the automaton $\mathcal{A}''$, after some work with the copy, would also return to the same stack configuration in state $f(q)$. Hence $\mathcal{A}$ accepts the same words as $\mathcal{A}'$.

Observe that the automaton $\mathcal{A}$ never increases and then decreases the number of stacks, without reading any letter in between (as in such situation it makes the „shortcut" described above). When a word $\sigma(w)$ is read, the decrease can not happen also after reading a letter (from Lemma 4). Hence $\mathcal{A}$ never decreases the number of stacks while reading $\sigma(w)$. ◀

The next lemma says that the automaton can know at each moment if the word read already is a prefix of a bracket expression or not. To formalize this, we replace the read_0 operation by two operations: read_0^{PBE} and read_0^{bad}.

▶ **Lemma 6.** *Assume there exists a D2PDA recognizing U. Then there exists a D2PDA $\mathcal{A}$ recognizing U, which instead of read_0 operation uses read_0^{PBE} if the word already read is a prefix of a bracket expression, and read_0^{bad} otherwise. Moreover, there exists a constant k and a k-filling σ such that $\mathcal{A}$ reading $\sigma(w)$ for some $w \in PBE$ never decreases the number of stacks.*

Proof. The constant k and the filling σ is taken from Lemma 5. Let $\mathcal{A}'$ be the automaton from Lemma 5; we will improve it, getting an automaton $\mathcal{A}$. We enrich the input alphabet by a # symbol and we consider the language

$$U' = U \cup \{w\# : w \in PBE\}.$$

We construct first a D2PDA $\mathcal{B}$ recognizing U'. Observe that $w \in PBE$ if $w \in \{[,]\}^*$ and $w*^k \in U$ for some k. Of course $\mathcal{B}$ in its state can remember if the input contained only brackets. Hence, after a # is read, it is enough to check if, after reading some number of stars, the automaton $\mathcal{A}'$ would accept (additionally, when something appears after the # symbol, $\mathcal{B}$ can not accept). It is easy to do so. We make a copy of $\mathcal{A}'$, in which instead of doing a read_0 operation, we assume that a star was read. When $\mathcal{A}'$ does read_{acc}, we accept our word.

An automaton $\mathcal{C}$ (also recognizing U') is constructed using a trick like in the previous lemma. Together with each stack symbol we remember a function $f\colon Q \to Q \cup \{\mathsf{acc}, \mathsf{na}\}$, where Q is the set of states of $\mathcal{B}$. It is defined in the same way as in the proof of the previous lemma, but it distinguishes an accepting and a non accepting read operation: when a run from q leads to a read_{acc} operation, we assign $f(q) = \mathsf{acc}$, and when it leads to read_0 (or the run is infinite), we assign $f(q) = \mathsf{na}$. The automaton can put on the stack the correct function together with each symbol. (One may ask if it is possible that $f(q) \in Q$, i.e. that the automaton decreases the number of stacks. It is possible, because it does not decrease the number of stacks only while reading the filling; here we can read arbitrary words, in particular containing $*$ or # symbols.)

Moreover on the top of each stack except the last we keep a function $g\colon Q \to \{\mathsf{acc}, \mathsf{na}\}$; for the last stack the function is kept in the state of $\mathcal{C}$. The function for an s-th stack is defined in the following way: Assume that there are only the first $s-1$ stacks; start a run

of $\mathcal{B}$ from a state q and continue it to the first read operation. If it is the read_{acc} operation, we take $g(q) = \mathsf{acc}$, otherwise $g(q) = \mathsf{na}$. Notice that g for an s-th stack depends only on g for the $s-1$-th stack (which „describes" first $s-2$ stacks) and on f on the top of the $s-1$-th stack (which „describes" the $s-1$-th stack). Hence g can be computed whenever a copy operation is done.

Finally we construct $\mathcal{A}$. It works like $\mathcal{C}$, but when a read_0 operation is going to be done, we look at f at the current character and g for the current stack. Assume reading a # character would end in a state q. If $f(q) = \mathsf{acc}$ we make the read_0^{PBE} operation, if $f(q) = \mathsf{na}$ we make the read_0^{bad} operation. Otherwise $f(q)$ is a state; if $g(f(q)) = \mathsf{acc}$ we make the read_0^{PBE} operation, if $g(f(q)) = \mathsf{na}$ we make the read_0^{bad} operation. Note that $\mathcal{A}$ still recognizes U'. Hence when the input alphabet is limited to $\{[,],*\}$, it recognizes U. Moreover, it uses the operation read_0^{PBE} after a word w, when $w\# \in U'$, hence when w is a prefix of a bracket expression. ◀

For the rest of the proof fix the automaton $\mathcal{A}$, the constant k and the k-filling σ, which are the result of Lemma 6.

For any number $n \geq 1$, let

$$w_n = \underbrace{[^{n+1}]^n[^{n+1}]^n \dots [^{n+1}]^n}_{|Q|+1 \text{ times}}.$$

We will see that after reading a word $\sigma(w_n)$, the number of symbols on the last stack has to be small.

▶ **Lemma 7.** *There exists a constant H such that for any $n \geq 1$ after reading the word $\sigma(w_n)$ the number of symbols on the last stack of $\mathcal{A}$ is not greater than H.*

Proof. For each prefix u of the word $\sigma(w_n)$ we define a block number: u is in the first block if u is a prefix of $\sigma([^{n+1})$, in the second block if u is a prefix of $\sigma([^{n+1}]^n)$ but not of $\sigma([^{n+1})$, in the third block if u is a prefix of $\sigma([^{n+1}]^n[^{n+1})$ but not of $\sigma([^{n+1}]^n)$, etc. Observe the following property $\star\star$. Consider two prefixes u_1 and u_2 of $\sigma(w_n)$ in the same block such that $|u_2| \geq |u_1| + (a+2k)(k+1)$ for some $a \geq 0$. We have

$$\begin{aligned} &open(u_2) \geq open(u_1) + a && \text{if the block number is odd,} \\ &open(u_2) \leq open(u_1) - a && \text{if the block number is even.} \end{aligned}$$

Indeed, assume the block number is odd. Consider the word $u_1^{-1}u_2$ (the suffix of u_2 which is after u_1). At the beginning it contains a suffix of a bracket expression (up to k letters), then opening brackets (coming from w_n) alternating with short (up to k letters) bracket expressions, and finally a prefix of a bracket expression (up to k letters). There are at least $a+2k$ opening brackets coming from w_n (as $|u_2| \geq |u_1| + (a+2k)(k+1)$). In the bracket expressions the number of opening and closing brackets is the same. In the initial fragment the balance is violated by at most k; the same for the final fragment.[1] Thus $open(u_2) \geq open(u_1) + a$. For even block number (having closing brackets) we get the opposite inequality.

Now come to a proof of the lemma. It is important that the automaton never decreases the number of stacks (thesis of Lemma 6). Hence, as long as it reads brackets, it can access only symbols on the last stack. Consider the run reading some $\sigma(w_n)$; assume its

[1] In fact, only the prefix or the suffix mattes, not both of them, so we could replace $a+2k$ by $a+k$.

configurations are numbered from 1 to some l, and in the last configuration the number of symbols on the last stack is h. From the last configuration the automaton tries to do the read_0^{PBE} operation. For each i ($1 \leq i \leq h$) let $p(i) - 1$ denote the number of the last configuration in the run such that the number of symbols on the last stack is smaller than i. Hence in the operation leading to configuration $p(i)$ the i-th symbol is pushed on the last stack and later it is never poped. To each i ($1 \leq i \leq h$) we assign a triple (x, q, γ), where $1 \leq x \leq 2(|Q| + 1)$, $q \in Q$, $\gamma \in \Gamma$. Here x is the block number of the prefix already read in configuration $p(i)$, q is the state in configuration $p(i)$, and γ is the stack symbol on position i on the last stack (in all moments between $p(i)$ and l).

There is a constant H (depending on k and $|Q|$) such that whenever $h > H$, some triple (x, q, γ) has to repeat at least $(2k + |Q| + 2)(k + 1) + 1$ times. Assume first that x in this triple is even (i.e. it corresponds to a block of closing brackets). Take any $c = (2k+1)(k+1)$ numbers $i_1 < i_2 < \cdots < i_c$ to which this triple is assigned. For each j, the run after $p(i_j)$ has no access to the symbols below i_j on the last stack (as well as to the symbols on the earlier stacks). Thus it depends only on the i_j-th stack symbol, the state, and the input word. Notice that the run between $p(i_j)$ and $p(i_{j+1})$ does at least one read operation, as otherwise the fragment between $p(i_j)$ and $p(i_{j+1})$ would repeat forever (the automaton is deterministic). Let r be the number of read operations in the run between $p(i_1)$ and $p(i_c)$. We have $r \geq c - 1$. From $\star\star$ it follows that the part of the input returned by these r read operations contains more closing brackets than opening brackets. Let repeat $|Q| + 2$ more times the fragment of the run from $p(i_1)$ to $p(i_c)$ (precisely, we repeat the operations done in this fragment, together with the part of the input returned by the read operations, and we leave the operations done later). We get a correct run on a new word, in particular after the last configuration it also does the read_0^{PBE} operation. But the new input word is not a prefix of a bracket expression, as it has too many closing brackets. This contradicts with the assumption that our automaton satisfies the thesis of Lemma 6, i.e. that it should end now doing the read_0^{bad} operation.

The argument is similar for odd x, but we have to consider $c = (2k + |Q| + 2)(k + 1) + 1$ numbers $i_1 < \cdots < i_c$ to which the repeating triple (x, q, γ) is assigned. As previously, there is at least one read operation between $p(i_j)$ and $p(i_{j+1})$ for each j. Thus the number r of read operations between $p(i_1)$ and $p(i_c)$ is at least $c - 1$. This time we remove the fragment of the run from $p(i_1)$ to $p(i_c)$. From $\star\star$, the part of the input read between $p(i_1)$ and $p(i_c)$ contained at least $|Q| + 2$ more opening brackets than closing brackets. We get the same contradiction as previously, as the new word is not a prefix of a bracket expression. ◀

For any $n \geq 1$, $0 \leq c \leq |Q|$ we will define a number $d(n, c)$. Assume that after reading $\sigma(w_n)$ there are s stacks. Now see what happens when we read the word $\sigma(w_n]^c)*^\omega$, where $*^\omega$ means that we give infinitely many stars to the automaton and we look at the infinite run. We look for the first of the two situations:

1. the automaton accepts (i.e. makes a read_{acc} operation), or
2. the number of stacks goes below s.

Note that for sure the automaton accepts after some number of stars (but possibly the second situations appears earlier). Note also that none of these situations can appear before we start reading the stars: during reading $\sigma(v)$ for any $v \in PBE$ the number of stacks does not decrease, and no word without stars can be accepted. Let $d(n, c)$ be the number of stars after which the earlier of these two situations appears.

Observe that $d(n, c)$ depends only on the content of the last stack (stack s) after reading $\sigma(w_n)$, on the state in this moment, and on the suffix of the filling $\sigma(w_n]^c)$ which appears after $\sigma(w_n)$. This is because the run reading $\sigma(w_n]^c)*^\omega$ never accesses stacks below s,

until some of the two interesting situations appear. Hence there are only finitely many possibilities: The number of the suffixes is finite, as their length is bounded by $|Q|(k+1)$. Thanks to Lemma 7 after reading $\sigma(w_n)$ stack s has height not greater than H, so the number of its different contents is finite. Thus there is a common upper bound D for all $d(n,c)$.

Let now fix $n = D$. Let s be the number of stacks after reading $\sigma(w_n)$. We define a partial function $a\colon Q \to \mathbb{N}$. Let us remove the stack s and start the automaton from a state q on the input word $*^\omega$. If in this infinite run $\mathcal{A}$ makes the read_{acc} operation only once, then let $a(q)$ denote the number of stars after which this happens. In the other cases ($\mathcal{A}$ never accepts or accepts multiple times) $a(q)$ is undefined. Let $u_c = \sigma(w_n]^c)$ for $0 \le c \le |Q|$. Consider the run on some of the words $u_c{*}^{char(u_c)+1}$. Note that $char(u_c) \ge 2D+1 > D$ (we count at least the length of prefix $\sigma([^{n+1}]^n)$), hence after reading $d(n,c) \le D$ stars $\mathcal{A}$ can not accept. Thus the number of stacks becomes $s-1$. The rest of the run depends only on the state q in this moment (as the content of the first $s-1$ stacks is the same for each c); the read_{acc} operation will appear after $a(q)$ more stars (in particular $a(q)$ is defined for this q). Hence $char(u_c) - D \le a(q) \le char(u_c)$. As there are only $|Q|$ states, and $|Q|+1$ values of c, some state q has to be used for two values of c, say c_1 and c_2 ($c_1 < c_2$). Note that $char(u_{c_1}) \ge char(u_{c_2}) + 2D + 1 > char(u_{c_2}) + D$ as to $char(u_{c_1})$ we count at least two blocks of brackets more than to $char(u_{c_2})$. This is a contradiction, as

$$a(q) \le char(u_{c_2}) < char(u_{c_1}) - D \le a(q).$$

5 Future work

The following question remains open: is there a language recognized by a collapsible deterministic higher order pushdown automaton which is not recognized by any deterministic higher order pushdown automaton without collapse *of any level*? It is possible that the language U from Theorem 1 has this property.

References

1. Klaus Aehlig, Jolie G. de Miranda, and C.-H. Luke Ong. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In *TLCA*, pages 39–54, 2005.
2. Klaus Aehlig, Jolie G. de Miranda, and C.-H. Luke Ong. Safety is not a restriction at level 2 for string languages. In *FoSSaCS*, pages 490–504, 2005.
3. Didier Caucal. On infinite terms having a decidable monadic theory. In *MFCS*, pages 165–176, 2002.
4. Bruno Courcelle. The monadic second-order logic of graphs ix: machines and their behaviours. *Theor. Comput. Sci.*, 151(1):125–162, 1995.
5. Bruno Courcelle and Teodor Knapik. The evaluation of first-order substitution is monadic second-order compatible. *Theor. Comput. Sci.*, 281(1-2):177–206, 2002.
6. M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS '08*, pages 452–461, Washington, DC, USA, 2008. IEEE Computer Society.
7. Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS '02*, pages 205–222, London, UK, 2002. Springer-Verlag.
8. Teodor Knapik, Damian Niwinski, Pawel Urzyczyn, and Igor Walukiewicz. Unsafe grammars and panic automata. In *ICALP*, pages 1450–1461, 2005.

9 Naoki Kobayashi. Model-checking higher-order functions. In *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 25–36, New York, NY, USA, 2009. ACM.

10 A. N. Maslov. The hierarchy of indexed languages of an arbitrary level. *Soviet Math. Dokl.*, 15:1170–1174, 1974.

11 C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS '06*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.