Visit-Bounded Stack Automata

Jozef Jirásek(✉) and Ian McQuillan

Department of Computer Science, University of Saskatchewan,
Saskatoon, SK S7N 5A9, Canada
jirasek.jozef@usask.ca, mcquillan@cs.usask.ca

Abstract. An automaton is *k-visit-bounded* if during any computation its work tape head visits each tape cell at most k times. In this paper we consider stack automata which are k-visit-bounded for some integer k. This restriction resets the visits when popping (unlike similarly defined Turing machine restrictions) which we show allows the model to accept a proper superset of context-free languages and also a proper superset of languages of visit-bounded Turing machines. We study two variants of visit-bounded stack automata: one where only instructions that move the stack head downwards increase the number of visits of the destination cell, and another where any transition increases the number of visits. We prove that the two types of automata recognize the same languages. We then show that all languages recognized by visit-bounded stack automata are effectively semilinear, and hence are letter-equivalent to regular languages, which can be used to show other properties.

Keywords: Stack automata · Visit-bounded automata · Semilinear languages

1 Introduction

When introducing a machine model or a grammar system, one of the most useful properties is that of semilinearity. The idea of a language being semilinear is defined formally in Sect. 2, but equivalently, a language is semilinear if and only if it has the same Parikh image as some regular language [6]. In particular, when this property is effective for a machine model $\mathcal{M}$, there is a procedure to construct a letter-equivalent finite automaton from any such machine. It is well-known due to Parikh that the context-free languages have this property [12]. When this property is effective along with effective closure under homomorphism, inverse homomorphism, and intersection with regular languages (the full trio properties), it immediately implies several useful properties.

1. It provides a procedure to decide emptiness, finiteness, and membership [8].
2. The class can be augmented by reversal-bounded counters and the resulting class is still semilinear [5]—more generally, the smallest full trio (or even full AFL) containing the languages accepted by $\mathcal{M}$ that is also closed under intersection with one-way nondeterministic reversal-bounded multicounter machines [8] is also semilinear. The resulting family has the positive decidable properties of (1).

© Springer Nature Switzerland AG 2022
V. Diekert and M. Volkov (Eds.): DLT 2022, LNCS 13257, pp. 189–200, 2022.
https://doi.org/10.1007/978-3-031-05578-2_15

3. All bounded languages accepted by $\mathcal{M}$ are so-called *bounded semilinear languages* [9], and they can all be accepted by a deterministic machine model, one-way deterministic reversal-bounded multicounter machines [9], where we can decide containment and equivalence of two machines.
4. Properties related to counting functions and slenderness (having at most k strings of each length) can be decided [10].

It is also one of the key properties of a class of grammars being mildly context-sensitive [11], which was developed to encompass the properties that are important for computational linguistics.

Stack automata are a generalization of pushdown automata with the ability to push and pop at the top of the stack, and an added ability to read the contents of the stack in a two-way read-only fashion [2]. They are quite powerful however and can accept non-semilinear languages [1,3]. Checking stack automata are stack automata that cannot pop, and cannot push after reading from the stack. Here, we consider a restriction on stack automata. Given a subset E of the stack instructions (push, pop, stay, move left, or right), a machine is k-visit$_E$-bounded if, during any computation, its stack head visits each tape cell while performing an instruction of E at most k times; and it is visit$_E$-bounded if it is k-visit$_E$-bounded for some k. We omit E if it contains all instructions.

Importantly in this definition, when a cell is popped, the count towards this bound disappears with it, and any new symbols pushed start with a count of zero. This makes the definition in some ways more general than had we defined Turing machines with a visit-bounded worktape. This type of model was studied by Greibach [4], who studied one-way input with a single Turing machine work tape which it can edit (precisely, Greibach defines the machines to be preloaded with a string from a language family such as the regular languages—but as we are restricting our study to regular languages, this preloading does not affect the capacity). Greibach showed that the languages accepted by finite-visit Turing machines are a semilinear subset of the checking stack languages.

Here we show that a stack language is visit-bounded if and only if it is visit$_E$-bounded where E only contains an instruction to move left. We then show that the family of languages accepted by visit-bounded stack automata only contain semilinear languages, in contrast to stack automata generally. Furthermore, they form a language family properly between the context-free and stack languages.

Lastly, we show that the class of languages of Turing machines with a finite-visit (or finite-crossing) restriction (and a one-way input tape) is properly contained in the class of languages of finite-visit stack automata (as the former does not contain all context-free languages), demonstrating the power of our model while still preserving semilinearity. This makes the family useful towards showing that other families are semilinear.

2 Preliminaries

We refer to [6,7] for an introduction to automata and formal language theory. An *alphabet* Σ is a finite set of *symbols*. A *string* over Σ is a finite sequence of

symbols from Σ. The set of all strings over Σ, including the empty string λ, is denoted by Σ^*. A *language* is a subset of Σ^*.

Let w be a string over $\Sigma = \{a_1, a_2, \ldots, a_n\}$. The *length* of w, denoted by $|w|$, is the number of characters in w, with $|\lambda| = 0$. For $a \in \Sigma$, the number of occurrences of the character a in the string w is denoted by $|w|_a$. The *Parikh image* of a string w, denoted $\Psi(w)$, is the vector $(|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_n})$. We note that two strings have the same Parikh image if one is a permutation of the other. For a language $L \subseteq \Sigma^*$, let $\Psi(L) = \{\Psi(w) \mid w \in L\}$. Two languages L_1 and L_2 are *letter-equivalent* if $\Psi(L_1) = \Psi(L_2)$. Equivalently, every string in L_1 is a permutation of some string in L_2, and vice versa.

A subset Q of $\mathbb{N}^m$ (m-tuples) is a *linear set* if there exist $\vec{v_0}, \vec{v_1}, \ldots, \vec{v_r} \in \mathbb{N}^m$ such that $Q = \{\vec{v_0} + i_1\vec{v_1} + \cdots + i_r\vec{v_r} \mid i_1, \ldots, i_r \in \mathbb{N}\}$. We call $\vec{v_0}$ the constant and $\vec{v_1}, \ldots, \vec{v_r}$ the periods. A finite union of linear sets is a *semilinear set*. A language $L \subseteq \Sigma^*$ is semilinear if $\Psi(L)$ is a semilinear set. It is known that a language L is semilinear if and only if there exists a regular language L' with $\Psi(L) = \Psi(L')$ [6]. For a family of languages accepted by a class of machines $\mathcal{M}$, we say that the family is *effectively semilinear* if there is an algorithm to always determine the constant and the periods for each linear set (or equivalently the letter-equivalent finite automaton). The following is a classical result in automata theory.

Theorem 1 (Parikh's Theorem [12]). *Let L be a context-free language. Then $\Psi(L)$ is a semilinear set.*

Let NFA be the class of nondeterministic finite automata and NPDA be the class of nondeterministic pushdown automata. Given a class of machines $\mathcal{M}$, let $\mathcal{L}(\mathcal{M})$ be the family of languages accepted by $\mathcal{M}$.

2.1 Stack Automata

A nondeterministic one-way *stack automaton* is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where:

- Q is the finite set of states,
- Σ and Γ are the input and work tape alphabets;
- Let $I = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}(x), \mathtt{pop} \mid x \in \Gamma\}$ be the *instruction set*, then:
- $\delta \subseteq Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\rhd\}) \times Q \times I$ is the transition relation,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of final states.

The special symbol $\rhd$ denotes the left end of the work tape, which is identified with the bottom of the stack.

We will define the contents of the stack slightly differently (but equivalently) from previous definitions in order to better capture the new restrictions. The work tape shall be represented as a series of pairs (x, i), denoting individual tape cells, where $x \in \Gamma \cup \{\rhd\}$ is the symbol written in this cell, and $i \in \mathbb{N}$ is the number of times the automaton has visited this cell. Note that the transition function of the automaton only has access to the symbols written on the tape, and the automaton can not inspect the visit counters of the cells.

A *configuration* of the automaton M is a triple (q, w, γ), where:

- $q \in Q$ is the current state,
- $w \in \Sigma^*$ is the input that is still to be read,
- $\gamma \in (\{\triangleright\} \times \mathbb{N})(\Gamma \times \mathbb{N})^* \lrcorner (\Gamma \times \mathbb{N})^*$ is the current content of the work tape. The special symbol $\lrcorner$ denotes the position of the tape head, which is scanning the cell immediately preceding this symbol.

Now let $E \subseteq \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$ be a set of *expensive* instructions. These are the instructions that are counted as visits to the tape cell. The automaton performs all instructions on the work tape as usual for stack automata. When an expensive instruction is performed, the number of visits of the tape cell under the head after the instruction is completed is increased by one.

We define the *move relation* $\vdash$ between configurations of M using a set of expensive instructions E as follows: For $\iota \in \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$, let the *cost* of ι be $\mathrm{c}(\iota) = 1$ if $\iota \in E$, and $\mathrm{c}(\iota) = 0$ if $\iota \notin E$. Then:

- $(p, aw, \alpha(x, i)\lrcorner\beta) \vdash (q, w, \alpha(x, i + \mathrm{c}(\mathtt{S}))\lrcorner\beta)$
  if $(p, a, x, q, \mathtt{S}) \in \delta$,
- $(p, aw, \alpha(x, i)(y, j)\lrcorner\beta) \vdash (q, w, \alpha(x, i + \mathrm{c}(\mathtt{L}))\lrcorner(y, j)\beta)$
  if $(p, a, x, q, \mathtt{L}) \in \delta$,
- $(p, aw, \alpha(x, i)\lrcorner(y, j)\beta) \vdash (q, w, \alpha(x, i)(y, j + \mathrm{c}(\mathtt{R}))\lrcorner\beta)$
  if $(p, a, x, q, \mathtt{R}) \in \delta$,
- $(p, aw, \alpha(x, i)\lrcorner) \vdash (q, w, \alpha(x, i)(y, \mathrm{c}(\mathtt{push}))\lrcorner)$
  if $(p, a, x, q, \mathtt{push}(y)) \in \delta$, and
- $(p, aw, \alpha(x, i)(y, j)\lrcorner) \vdash (q, w, \alpha(x, i + \mathrm{c}(\mathtt{pop}))\lrcorner)$
  if $(p, a, y, q, \mathtt{pop}) \in \delta$;

where $p, q \in Q$, $a \in \Sigma \cup \{\lambda\}$, $w \in \Sigma^*$, $x \in \Gamma \cup \{\triangleright\}$, $y \in \Gamma$, $i, j \in \mathbb{N}$, $\alpha \in \{\lambda\} \cup ((\triangleright \times \mathbb{N})(\Gamma \times \mathbb{N})^*)$, $\beta \in (\Gamma \times \mathbb{N})^*$, and the work tape string on both sides of the relation is well-formed (in particular, $x = \triangleright$ if and only if $\alpha = \lambda$). Let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$.

A *computation* of a stack automaton M on a string $w \in \Sigma^*$ is a sequence of configurations $c_0 \vdash c_1 \vdash \cdots \vdash c_n$, where $c_0 = (q_0, w, (\triangleright, 0)\lrcorner)$, and $c_n = (q_n, \lambda, \gamma_n)$. If $q_n \in F$, this computation is *accepting*. The automaton M *accepts* a string w if there exists an accepting computation of M on w. The *language accepted by* M, denoted by $L(M)$, is the set of all strings from Σ^* that M accepts.

Let SA be the class of all stack automata. A stack automaton is called a *non-erasing stack automaton* if it uses no $\mathtt{pop}$ instructions. A non-erasing stack automaton is called a *checking stack automaton* if it cannot push again after either a $\mathtt{L}$ or $\mathtt{R}$ instruction. The class of non-erasing stack automata is denoted by NESA, and checking stack automata by CSA.

For an integer k and a set of expensive instructions E, we say that a computation of a stack automaton M is *k-visit$_E$-bounded*, if the number of visits of every cell in every configuration in this computation is less than or equal to k. We say that M is *k-visit$_E$-bounded* if for every string $w \in L(M)$ the automaton M has a k-visit$_E$-bounded accepting computation on w. Finally, M is *visit$_E$-bounded* if there is a finite $k \in \mathbb{N}$ such that M is k-visit$_E$-bounded. If we leave

off the subscript E, it is assumed that $E = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$. Let $\mathsf{VISIT}_E(k)$ be the class of k-visit$_E$-bounded, VISIT_E be all visit$_E$-bounded machines, and again we leave off the subscript E if $E = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$. It is immediate that $\mathcal{L}(\mathsf{SA}) = \mathcal{L}(\mathsf{VISIT}_\emptyset)$.

Note the important distinguishing feature of the stack automaton model which sets it apart from known visit-bounded Turing machine models: whenever a tape cell is popped from the top of the stack, the number of visits of that cell is reset. Whenever a new cell is pushed to the top of the stack, this new cell begins with a visit count of 0 (or 1, if `push` is an expensive instruction). This allows a visit-bounded stack automaton to perform some computations that an analogous visit-bounded Turing machine could not.

3 Visit-Bounded Automata

As we have seen in definitions in Sect. 2, the notion of a visit-bounded stack automaton is dependent on the choice of the set of expensive instructions E which increase the visit counters of tape cells. To begin, we consider two expensive instruction sets: $E = \{\mathtt{L}\}$, and $E = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$. In the first case, only `L` instructions increase the visit counters. In the second case, all instructions increase the visit counters.

Example 2. Let $M = (\{q_0\}, \{a\}, \{\}, \{(q_0, a, \triangleright, q_0, \mathtt{S})\}, q_0, \{q_0\})$ be a stack automaton. This simple automaton scans its input consisting of a number of symbols a, while the work tape head rests on the bottom of the stack marker.

Observe that M is visit$_{\{\mathtt{L}\}}$-bounded, as it never performs an `L` instruction, and thus the number of visits of the only used tape cell never increases above 0. On the other hand, M is not visit-bounded, as the `S` instructions in the only computation of M on string a^k increases the visit counter of the tape cell to k.

Every visit-bounded automaton is also visit$_{\{\mathtt{L}\}}$-bounded. Indeed, the number of visits to a cell can not increase if we only consider a limited subset of expensive instructions. Perhaps surprisingly, as we will show in Theorem 3, the converse is also true if we only consider languages accepted by the automaton. For any visit$_{\{\mathtt{L}\}}$-bounded automaton A, we can construct a visit-bounded automaton B with $L(B) = L(A)$. Therefore, limiting the usage of any instruction other than `L` does not reduce the descriptive power of the automaton model.

Theorem 3. *Let $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a visit$_{\{\mathtt{L}\}}$-bounded stack automaton. Then there exists a visit-bounded stack automaton B such that $L(B) = L(A)$. Hence, $\mathcal{L}(\mathsf{VISIT}) = \mathcal{L}(\mathsf{VISIT}_{\{\mathtt{L}\}})$.*

Proof. Let A be visit$_{\{\mathtt{L}\}}$-bounded, *i.e.*, A visits every tape cell using the `L` instruction at most k times. We prove the theorem by describing a construction of the automaton B. The basic idea of the construction is that B emulates a computation of A, but every symbol on the work tape of A shall be represented by multiple copies of the same symbol on the work tape of B. Instructions of A operating on a specific tape cell will be distributed among the copies of this cell

by B in such a way that every copy is only visited a fixed number of times. By careful counting we show that any computation of A can be emulated by B in such a way that the number of visits to every cell of B on any instruction can be bounded as a function of k. This means that there is a constant ℓ which depends on k such that B is ℓ-visit-bounded, *i.e.*, B is visit-bounded.

The detailed construction appeared in Appendix A of the submitted paper. ☐

As a consequence of Theorem 3, the classes of languages accepted by $\text{visit}_{\{\mathtt{L}\}}$-bounded and visit-bounded automata are identical.

We can also observe the following result for context-free languages:

Corollary 4. *For all* $E \subseteq \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$, $\mathcal{L}(\mathsf{NPDA}) \subsetneq \mathcal{L}(\mathsf{VISIT}_E)$.

Proof. A pushdown automaton can be seen as a stack automaton which never uses the L and R instructions. This automaton is trivially $\text{visit}_{\{\mathtt{L}\}}$-bounded, and by Theorem 3 its language can be accepted by some visit-bounded stack automaton. Strictness can be seen using $\{a^n b^n c^n \mid n > 0\}$. ☐

We conclude this section with a comparison to Turing machines. Consider nondeterministic Turing machines with a one-way read-only input and a single work tape. If there is a bound on the number of changes of direction on the work tape (reversal-bounded), we denote these machines by TMRB; if there is a bound on the number of times the boundary of each pair of adjacent cells is crossed (finite-crossing), we denote these machines by TMFC; and if there is a bound on the number of visits to each cell (finite-visit), we denoted these by TMFV. Greibach studies these machines [4] where the work tape is preloaded with regular languages (or other families but we do not consider others), and the work tape is confined to the preloaded space. This preloading does not impact the languages accepted however as shown in the proof of the following, along with a comparison to visit-bounded stack automata.

Proposition 5. $\mathcal{L}(\mathsf{TMRB}) \subsetneq \mathcal{L}(\mathsf{TMFC}) = \mathcal{L}(\mathsf{TMFV}) \subsetneq \mathcal{L}(\mathsf{VISIT})$.

Proof. First we will argue that preloading these Turing machines with regular languages does note affect the languages accepted. Indeed, preloading can be simulated by guessing and writing a preloaded string and then simulating. In the other direction, a new dummy symbol B can be introduced, and the machine can be preloaded with B^*, the machine then guesses some start position and simulates using B as the blank symbol. It will only accept if it is preloaded with a string that is longer than the number of cells visited and it guesses the correct start position. Greibach shows that $\mathcal{L}(\mathsf{TMRB}) \subsetneq \mathcal{L}(\mathsf{TMFC}) = \mathcal{L}(\mathsf{TMFV})$ in Theorems 2.15 and 3.12. To show that $\mathcal{L}(\mathsf{TMFV}) \subseteq \mathcal{L}(\mathsf{VISIT})$, in Lemma 4.21 of [4], Greibach shows that every $\text{Ł} \in \mathcal{L}(\mathsf{TMFV})$ can be accepted by a Turing machine preset with a regular language where the machine does not ever change the work tape contents, and every accepting computation is k-visit-bounded. Such a machine M can be accepted by a k-visit-bounded stack automaton by first guessing the stack contents, and then simulating. The inclusion is strict as

noted in the proof of Theorem 4.26 [4] as the context-free Dyck language cannot be accepted by a TMFV. □

4 Semilinearity

The main result of this section is to prove that the language accepted by any visit-bounded stack automaton is semilinear. To prove this, we give a procedure that, given a visit-bounded stack automaton M, constructs a pushdown automaton P, such that $L(P)$ and $L(M)$ are letter-equivalent. Specifically, we show that for any string $w \in L(M)$, the automaton P can accept some permutation of w, and vice versa. It is known that languages of pushdown automata are semilinear, and semilinearity is preserved under letter-equivalence, hence this proves the main result.

Theorem 6. *Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a visit-bounded stack automaton. Then the language accepted by M is effectively semilinear.*

Proof. Let M be k-visit-bounded for an integer k. Further, assume that the automaton ends its computation with an empty stack. If it does not, this can be achieved by deleting the entire content of the stack before accepting, which adds at most one visit to every tape cell.

The central concept used for the proof is the *visit history* of a tape cell. Using the analogy of a physical work tape with paper cells, every time the automaton makes a move, it records the transition it has just used (a 5-tuple (p, a, x, q, ι)) on both the cell it left and the cell it entered. If the transition used an `S` instruction, those two refer to the same cell. In this way, since every cell is visited at most k times before being destroyed, the visit history of every cell contains at most $2k$ entries: k for transitions which were used to enter the cell, and another k for transitions which were used to leave. We shall refer to the i-th entering transition as $t_{\text{in}}[i]$ and the i-th leaving transition as $t_{\text{out}}[i]$. Also note that throughout the computation of the machine every transition used is recorded exactly twice: once in the cell it begins in, and once in the cell it ends in. This connection links the visit histories of all cells into a linked list-like structure which records the entire computation of M. Since every transition contains the input symbol being read (if any), following these links allows us to see the string being accepted.

The main idea is to construct a pushdown automaton P, which emulates the `push` and `pop` instructions in some computation of M, while nondeterministically guessing the entire history of every cell pushed on the stack. As long as P can ensure the integrity of links between every pair of adjacent cells, the entire linked list can be followed to reconstruct a computation of M, including the `L`, `R`, and `S` instructions. Then if P also reads all input symbols corresponding to every t_{in} transition in all histories, it accepts a permutation of the string accepted by M in this computation.

An important fact affecting the construction of P is that cells on the work tape of M can be erased and replaced by another cell. Therefore, not all of the transitions in the history of one cell need to correspond to transitions in the

history of one adjacent cell. Some transitions could connect to a cell that had been in that place but was previously erased, and some transitions might connect to a cell that will be in that place in the future, after the currently following cell is erased. Therefore, the representation of every cell in P will additionally carry a *completed transition counter*, an index *ctc* in the range $1 \leq ctc \leq k$, which indicates how many transitions in the history of the current cell have already been matched with corresponding transitions in the histories of adjacent cells.

We can now describe the construction of the pushdown automaton P.

Definition 7. *A* history card *is a* $(2k+2)$*-tuple* $(x, ctc, t_{in}[1], \ldots, t_{in}[k], t_{out}[1], \ldots, t_{out}[k])$*, where:*

- $x \in (\Gamma \cup \{\triangleright\})$ *is the stack symbol written on the tape cell,*
- $1 \leq ctc \leq k$ *is the completed transition counter,*
- $t_{in}[i] \in (\delta \cup \{\emptyset\})$*, for* $1 \leq i \leq k$*, are the transitions ending in this cell, and*
- $t_{out}[j] \in (\delta \cup \{\emptyset\})$*, for* $1 \leq j \leq k$*, are the transitions originating in this cell.*

Not all possible history cards can appear in some computation of M. We impose several consistency constraints on the history cards that P can use, to ensure that the information on each card is filled in properly and does not contradict itself.

Definition 8. *A history card is* internally consistent*, if all the following hold:*

- $t_{in}[i] \neq \emptyset \iff t_{out}[i] \neq \emptyset$ *for all* $1 \leq i \leq k$*. If there is an incoming transition, there has to be a corresponding outgoing transition.*
- *If* $t_{in}[i] = \emptyset$*, then also* $t_{in}[i+1] = \emptyset$*. Similarly if* $t_{out}[i] = \emptyset$*, then also* $t_{out}[i+1] = \emptyset$*. This holds for all* $1 \leq i < k$*. Transitions are always stored in a contiguous block of indices starting from the beginning of the card.*
- *The transition* $t_{in}[1]$ *performs the* `push`(x) *instruction, where* x *is the symbol stored on this card. The last non-empty* $t_{out}[i]$ *performs the* `pop` *instruction. No other* t_{in} *transitions are* `push` *and no other* t_{out} *transitions are* `pop` *instructions. The history of a cell begins when it is pushed and ends when it is popped from the stack. Each of these events can only happen once in the lifetime of the cell.*
    - *The exception to the three rules above is a card with* $x = \triangleright$*. This card represents the bottom of the stack of* M*, and here the computation of* M *begins and ends. Therefore* $t_{in}[1] = \emptyset$*, there is exactly one* i *such that* $t_{in}[i] \neq \emptyset$ *and* $t_{out}[i] = \emptyset$*, no* t_{in} *is a* `push` *or* `R` *instruction, and no* t_{out} *is a* `pop` *or* `L` *instruction.*
- *The work tape symbol read in every* t_{out} *transition is the symbol* x *on this card.*

Denote by H the set of all internally consistent history cards. Note that $|H| \leq (|\Gamma| + 1)k(|\delta| + 1)^{2k}$. The set H shall be the working alphabet of the pushdown automaton P. An example of history cards and links between them corresponding to a computation of M is shown in Fig. 1. The links are not explicitly stored but will be implied.

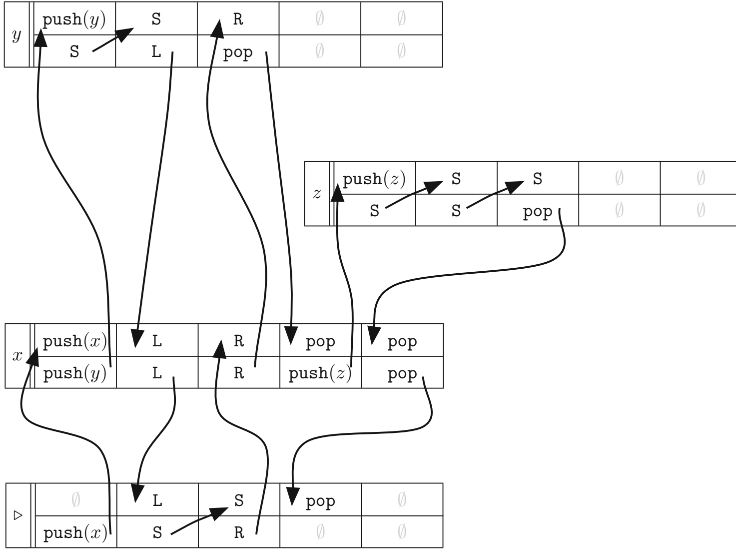

Fig. 1. Histories of tape cells after executing the following sequence of instructions: push(x), push(y), S, L, L, S, R, R, pop, push(z), S, S, pop, pop. Only the instructions used in the transitions are shown, states and symbols read are omitted. Transitions t_{in} shown in the top row, and t_{out} in the bottom row. Arrows show links between history cards formed by pairs of identical transitions.

Now we describe an algorithm used by P to simulate a computation of M. This algorithm employs two subprocedures: the first one advances the completed transition counter on a card step by step, verifying that the transitions on the card can link together to form a continuous computation, until either a push or a pop transition, or the end of the computation is reached. The facts that need to be verified are that S instructions on this card link to each other, and that every outgoing L instruction is followed by an incoming R instruction.

The second procedure takes two history cards as input and attempts to link together transitions between them. An outgoing push instruction on the bottom card has to link to the first incoming instruction on the top card. Every outgoing R instruction on the bottom card has to link to an incoming R instruction on the top card, and every outgoing L instruction on the top card has to link to an incoming L instruction on the bottom card. Finally, the last transition of the top card, performing a pop instruction, has to link to an incoming pop transition on the bottom card.

The complete description of both procedures appeared in Appendix B of the submitted paper. The important fact is that since there are only finitely many different history cards, the pushdown automaton itself does not have to perform either of these procedures. The results for all possible inputs can be encoded into its transition function.

A description of the algorithm performed by P is in Algorithm 4.1.

1 Nondeterministically choose a history card containing the symbol ▷. Push this card on the stack.
2 Read all input symbols that are read in any incoming transition on this card.
3 **while** *There is a history card on the stack* **do**
4   Advance the *ctc* of the card on top of the stack, verifying the consistency of instruction links, until either a `push` or a `pop` instruction, or the end of the computation is encountered.
5   **if** *The transition encountered performs a* `push` *instruction* **then**
6   Nondeterministically choose a history card containing the symbol being pushed.
7   Verify that the chosen card can be matched to the card currently on top of the stack.
8   **if** *The cards can be matched together* **then**
9   Move the *ctc* of the card on top of the stack to the incoming `pop` instruction corresponding to the removal of the cell represented by the new card.
10   Push the newly chosen card on top of the stack, initializing its *ctc* to 1.
11   Read all input symbols that are read in any incoming transition on the new card.
12   **else**
13   Halt the computation and reject.
14   **end**
15   **else if** *The transition encountered performs the* `pop` *instruction, or the end of the computation is encountered* **then**
16   Erase the top card from the stack.
17   **else if** *A transition on the card can not be linked properly* **then**
18   Halt the computation and reject.
19   **end**
20 **end**
21 Finish the computation and accept.

Algorithm 4.1: The algorithm performed by the pushdown automaton P emulating a computation of a visit-bounded stack automaton.

If the computation of P succeeds, this means that all transitions in all the history cards used can be linked together to form one possible contiguous computation of M. Further, P reads every symbol that is read by every instruction in this computation, just not necessarily in the same order as M. However, this means that the string read by P is a permutation of the string that is read by the corresponding computation of M. Therefore, the language of P is letter-equivalent to the language of M. Finally, since the languages of pushdown automata are semilinear, and semilinearity is preserved under letter-equivalence, this means that the language of M is semilinear as well. □

5 Other Expensive Instruction Sets

We have considered automata models with $E = \{\mathtt{L}\}$ and $E = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$. We can ask whether models with other expensive instruction sets also describe the same class of languages.

It is possible to show that $\text{visit}_{\{\mathtt{R}\}}$-bounded automata accept the same class of languages as $\text{visit}_{\{\mathtt{L}\}}$-bounded automata. The proof uses similar ideas as in the construction in the proof of Theorem 3, though we do not include it here. Adding the S instruction to a set of expensive instructions does not change the class of languages accepted, as every S instruction can be replaced by a pair of R and L instructions, or push and pop instructions when operating on top of the stack. Therefore it is always possible to construct an equivalent automaton which never uses the S instruction.

Hence, if we consider expensive instruction sets E containing either L or R, any visit-bounded automaton is also visit_E bounded for such E. Therefore all models with such an expensive instruction set accept the same class of languages.

Making expensive instructions exactly the push instructions has no effect on the languages accepted (i.e. it accepts all stack languages), as any cell can only be pushed on the stack once. We only include the push instruction as a possible expensive instruction for completeness.

Finally, we shall see that a model with $E = \{\mathtt{pop}\}$ also accepts all stack automaton languages. Using a procedure similar to the one in the construction of automaton B in Theorem 3 we can clone symbols on the stack and replace every pop transition by a sequence $\mathtt{pop} - \mathtt{pop} - \mathtt{push}$, such that every cell is visited at most twice by pop instructions.

These results can be summarized as follows. The hierarchy is depicted in Fig. 2.

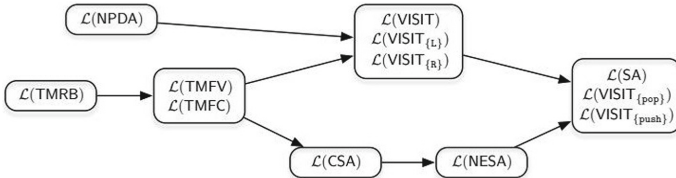

Fig. 2. The language families listed are related such that the families that are equal are written together in a box, inclusions are shown with an arrow that are proper in every case, and no lines connecting them indicate that they are incomparable.

Theorem 9. *The hierarchy shown in Fig. 2 is correct.*

Proof. That $\mathcal{L}(\mathsf{TMRB}) \subsetneq \mathcal{L}(\mathsf{TMFC}) = \mathcal{L}(\mathsf{TMFV}) \subsetneq \mathcal{L}(\mathsf{VISIT})$ is shown in Proposition 5. That $\mathcal{L}(\mathsf{TMFV}) \subsetneq \mathcal{L}(\mathsf{CSA})$ is shown in [4]. That $\mathcal{L}(\mathsf{CSA}) \subsetneq \mathcal{L}(\mathsf{NESA}) \subsetneq \mathcal{L}(\mathsf{SA})$ is well known [3]. That $\mathcal{L}(\mathsf{NPDA}) \subsetneq \mathcal{L}(\mathsf{VISIT})$ was shown in Corollary 4. That $\mathcal{L}(\mathsf{VISIT}) = \mathcal{L}(\mathsf{VISIT}_{\{\mathtt{L}\}})$ is from Theorem 3, and the equality with $\mathcal{L}(\mathsf{VISIT}_{\{\mathtt{R}\}})$ is mentioned above. The equality of $\mathcal{L}(\mathsf{SA})$ with $\mathcal{L}(\mathsf{VISIT}_{\{\mathtt{pop}\}})$ and

$\mathcal{L}(\mathsf{VISIT}_{\{\mathtt{push}\}})$ is also mentioned above. The proper inclusion of $\mathcal{L}(\mathsf{VISIT})$ in $\mathcal{L}(\mathsf{SA})$ follows from Theorem 6 since stack automata can accept non-semilinear languages [3]. Also, $\mathcal{L}(\mathsf{CSA})$ contains languages not accepted by $\mathcal{L}(\mathsf{VISIT})$ since $\mathcal{L}(\mathsf{CSA})$ contains non-semilinear languages [3]. Also, it is known that $\mathcal{L}(\mathsf{NESA})$ does not contain all context-free languages [4]. □

Hence, all the families above that are semilinear are contained in $\mathcal{L}(\mathsf{VISIT})$, making it the most powerful such family.

References

1. Ginsburg, S., Greibach, S., Harrison, M.: One-way stack automata. J. ACM **14**(2), 389–418 (1967)
2. Ginsburg, S., Greibach, S., Harrison, M.: Stack automata and compiling. J. ACM **14**(1), 172–201 (1967)
3. Greibach, S.: Checking automata and one-way stack languages. J. Comput. Syst. Sci. **3**(2), 196–217 (1969)
4. Greibach, S.A.: One way finite visit automata. Theor. Comput. Sci. **6**, 175–221 (1978)
5. Harju, T., Ibarra, O., Karhumäki, J., Salomaa, A.: Some decision problems concerning semilinearity and commutation. J. Comput. Syst. Sci. **65**(2), 278–294 (2002)
6. Harrison, M.: Introduction to Formal Language Theory. Addison-Wesley, Reading (1978)
7. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
8. Ibarra, O., McQuillan, I.: Semilinearity of families of languages. Int. J. Found. Comput. Sci. **31**(8), 1179–1198 (2020)
9. Ibarra, O.H., McQuillan, I.: On families of full trios containing counter machine languages. Theor. Comput. Sci. **799**, 71–93 (2019)
10. Ibarra, O.H., McQuillan, I., Ravikumar, B.: On counting functions and slenderness of languages. Theor. Comput. Sci. **777**, 356–378 (2019)
11. Joshi, A.K.: Tree adjoining grammars: how much context-sensitivity is required to provide reasonable structural descriptions? In: Natural Language Parsing, pp. 206–250. Cambridge University Press, Cambridge (1985)
12. Parikh, R.: On context-free languages. J. ACM **13**(4), 570–581 (1966)