View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Queen Mary Research Online

Reachability in Pushdown Register Automata☆

A. S. Murawski[a], S. J. Ramsay[a,*], N. Tzevelekos[b]

[a]*Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK*
[b]*School of Electronic Engineering and Computer Science, Queen Mary University of London, London E1 4NS, UK*

Abstract

We investigate reachability in pushdown automata over infinite alphabets. We show that, in terms of reachability/emptiness, these machines can be faithfully represented using only $3r$ elements of the alphabet, where r is the number of registers. We settle the complexity of associated reachability/emptiness problems. In contrast to register automata, the emptiness problem for pushdown register automata is EXPTIME-complete, independent of the register storage policy used. We also solve the global reachability problem by representing pushdown configurations with a special register automaton. Finally, we examine extensions of pushdown storage to higher orders and show that reachability is undecidable at order 2.

Keywords: register automata, pushdown automata, reachability, infinite alphabets, complexity

1. Introduction

Recent years have seen lively interest in automata over infinite alphabets. This has largely been driven by applications which, although diverse, had a common thread in dealing with alphabets of potentially unbounded size for which finite-domain abstractions were deemed unsatisfactory. A case in point are markup languages [20, 4], most notably XML. Here, documents contain data values whose range is potentially unbounded and queries are allowed to perform comparison tests on such data. A similar scenario occurs in reference-based programming languages, such as object-oriented [6, 2, 12, 17] or ML-like languages [18, 19]. In such languages, memory is managed with the help of reference names that can be created afresh and compared for equality but are otherwise abstract. Other examples include array-accessing programs [1], which are allowed to use the array to store and compare elements of an unbounded domain, as well as programs with restricted integer parameters [7].

Such applications call for a robust theory of automata over infinite alphabets which can lead to an understanding comparable to that of the finite-alphabet setting. Such a theory will expose the limits of this model of computation and establish a complexity-theoretic guide for applications. A lot of the groundwork, surveyed in [22, 3], was already

☆Research supported by the Engineering and Physical Sciences Research Council (EP/J019577/1) and the Royal Academy of Engineering (Tzevelekos, RF).
*Corresponding author. (Phone: +441386422040).
Email addresses: a.murawski@warwick.ac.uk (A. S. Murawski), s.ramsay@warwick.ac.uk (S. J. Ramsay), nikos.tzevelekos@qmul.ac.uk (N. Tzevelekos)

dedicated to uncovering a notion of "regularity" in the infinite-alphabet case. Very early in that work, a way was proposed to extend the concept of finite memory to infinite alphabets which consisted of introducing a fixed number of *registers* for storing elements of the alphabet [13]. The constructed automata, called *Finite-Memory Automata* [13] or *Register Automata* [20], would otherwise look just as finite-state automata where the transition labels would also involve indices referring to register addresses. Building on the success of this work, another strand aimed to identify the infinite-alphabet "context-free" languages. To this end, Cheng and Kaminski [8] introduced a notion of context-free grammars over infinite alphabets and defined a corresponding notion of pushdown automata. Later, Segoufin presented a similar definition in [22], albeit couched in a way suitable to process data words.

Our paper is devoted to studying exactly such context-free computational scenarios through an investigation of *pushdown register systems* (PDRS), devices in which registers are integrated with a pushdown store. Although of foundational nature, the work is largely motivated by the pertinence of such machines to software model checking [6, 2]. A particular such application is in game-semantics-based verification [19, 16], whereby the semantics of programs is algorithmically given by means of variants of pushdown register systems, which in turn are used as models to be fed in procedures for checking program equivalence. We present several new results on the complexity of reachability testing, which altogether fill a gap in the theory of "context-free" languages over infinite alphabets. More specifically, we make the following contributions.

Alphabet distinguishability. A finite-memory automaton [13] with r registers can store r elements of the infinite alphabet at any instant. In fact, such automata are only capable of remembering r elements of the infinite alphabet over the course of a run — for any accepting run one can construct another one involving only r elements of the alphabet.[1] Here we show that, even though pushdown register systems have no bound on the number of elements of the alphabet that they can store at any instant, over the course of a run they can really "remember" at most $3r$ of them. More precisely, we show that for any run of a PDRS with r registers there exists an equivalent run, i.e. with the same initial and final configurations, but in which every configuration contains register assignments drawn from only $3r$ elements. Moreover, no smaller number is enough: we exhibit a family of PDRS whose runs require remembering at least $3r$ elements.

Reachability testing. The above-mentioned result yields an obvious methodology for reductions to the finite-alphabet setting, which immediately implies decidability of associated reachability and language emptiness problems. While the decidability of emptiness has already been proved in [8] using context-free grammars, we provide exact complexity bounds for the problem, namely, EXPTIME-completeness.

In the pushdown-free setting, language nonemptiness was known to be NL-, NP- and PSPACE-complete, depending on the register discipline. Three register assignment disciplines have been studied in the literature, which we shall call *single and full* (SF), *single and initially empty* ($S\#_0$) and *multiple and full* (MF). A single assignment discipline

[1] For register automata, the corresponding bound is $r + 1$.

Register Discipline	SF	$S\#_0$	MF
RA	NL-c	NP-c	PSPACE-c
PDRA	EXPTIME-c	EXPTIME-c	EXPTIME-c

Table 1: Complexity of the emptiness problem.

requires that the contents of all registers be distinct, whereas a multiple assignment discipline allows for duplicate register contents. A full assignment discipline, on the other hand, requires that at all times every register must contain some letter from the infinite alphabet, whereas an initially empty discipline allows registers to be empty at the start of a run. The complexity of emptiness for register automata according to the assumed register discipline is given in the first row of Table 1: in the SF case the problem is NL-complete, as it coincides with emptiness of the underlying finite-state automaton; in the $S\#_0$ case it becomes NP-complete [21], as one is able to use the registers to encode boolean assignments; while in the MF case one is able to encode a linear-size tape with the registers, and therefore the problem is PSPACE-complete [10]. In contrast, in the pushdown case, we show that such distinctions do not affect the complexity: even if identical elements can be kept in different registers, the problem can still be solved in EXPTIME and it is EXPTIME-hard already in the case where only distinct elements are allowed. In the latter case, the hardness proof is technically involved since sequences of distinct names do not provide a supportive framework for representing memory content (as needed in reduction arguments using computation histories).

Global reachability. We give a simple, exponential-time algorithm for *global* reachability analysis. This analysis asks for a representation of all configurations from which a specified set of target configurations can be reached. In the finite-alphabet case it is well known that, if the target set is regular, the set of configurations that reach it can be captured by a finite automaton [5]. We prove an analogous result in the infinite-alphabet setting: given a PDRS $\mathcal{S}$ and a register automaton defining a set of target configurations T of $\mathcal{S}$, our algorithm constructs a new register automaton that represents exactly the set of configurations that can reach some configuration in T. To ensure the algorithm is a smooth analogy of the saturation algorithm of [5], it manipulates a particularly succinct variant of register automata which we call a *register manipulating register automaton* (RMRA), but we show that such machines can always be transformed to a machine of the usual kind for at most an exponential increase in size.

Higher-order. Higher-order pushdown automata [15] take the idea of pushdown storage further by allowing for nesting. Standard pushdown store is considered to be order 1, while the elements stored in an order-k ($k > 1$) pushdown store are $(k-1)$-pushdown stores. In the finite alphabet setting this leads to an infinite hierarchy of decidable models of computation with a $(k-1)$-EXPTIME-complete problem at order k. We examine how the model behaves in the infinite alphabet setting, after the addition of a fixed number of registers for storing elements of the infinite alphabet.

We first observe that one can no longer establish a uniform bound on the number of symbols of the infinite alphabet that suffice to represent arbitrary runs. The existence of such a bound would imply decidability of the associated

reachability problems, but the lack of a bound is not sufficient for establishing undecidability: indeed, the *decidable* class of data automata from [4] contains an automaton that can recognize all words consisting of distinct letters. Still, we show that the reachability problem for higher-order register pushdown automata is undecidable, already at order 2 and with one register.

2. Basic Definitions

Let us assume a countably infinite alphabet $\mathcal{D}$ of ***data values*** or ***names***. We introduce a simple formalism for computations based on a finite number of $\mathcal{D}$-valued registers and a pushdown store. Writing $[r]$ for $\{1, \cdots, r\}$, by an ***r-register assignment*** we mean an injective map from $[r]$ to $\mathcal{D}$. We write Reg_r for the set of all such assignments.[2]

Definition 1. A ***pushdown r-register system*** (r-PDRS) is a tuple $\mathcal{S} = \langle Q, q_I, \tau_I, \delta \rangle$, where:

- Q is a finite set of *states*, with $q_I \in Q$ being initial,
- $\tau_I \in Reg_r$ is the *initial r-register assignment*,
- and $\delta \subseteq Q \times Op_r \times Q$ is the *transition relation*,

with $Op_r = \{ i^\bullet, push(i), pop(i) \mid 1 \leq i \leq r \} \cup \{ pop^\bullet \}$.[3]

The operations executed in each transition have the following meaning: the $i^\bullet$ operation *refreshes* the content of the i-th register; $push(i)$ pushes the symbol currently in the i-th register on the stack; $pop(i)$ pops the stack if the top symbol is the same as that stored in the i-th register; $pop^\bullet$ pops the stack if the top of the stack is currently not present in any of the registers. This semantics is given formally below.

Definition 2. A ***configuration*** of an r-PDRS $\mathcal{S}$ is a triple $(q, \tau, s) \in Q \times Reg_r \times \mathcal{D}^*$. We say that (q_2, τ_2, s_2) is a *successor* of (q_1, τ_1, s_1), written $(q_1, \tau_1, s_1) \vdash (q_2, \tau_2, s_2)$, if $(q_1, op, q_2) \in \delta$ for some $op \in Op_r$ and one of the following conditions holds.

- $op = i^\bullet$, $\forall j.\, \tau_2(i) \neq \tau_1(j)$, $\forall j \neq i.\, \tau_2(j) = \tau_1(j)$ and $s_2 = s_1$.
- $op = push(i)$, $\tau_2 = \tau_1$ and $s_2 = \tau_1(i)s_1$.
- $op = pop(i)$, $\tau_2 = \tau_1$ and $\tau_1(i)s_2 = s_1$.
- $op = pop^\bullet$, $\tau_2 = \tau_1$ and, for some $d \in \mathcal{D}$, $\forall j.\, \tau_1(j) \neq d$ and $ds_2 = s_1$.

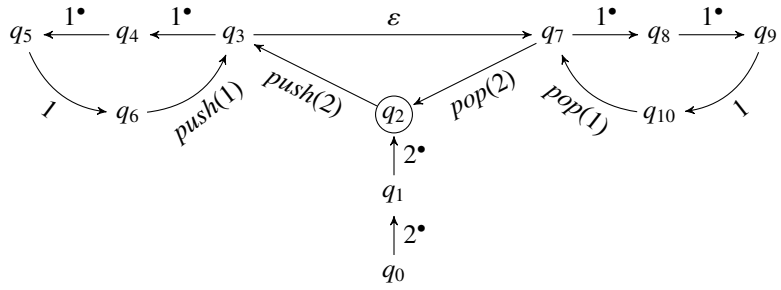
A ***transition sequence*** of $\mathcal{S}$ is a sequence $\rho = \kappa_0, \cdots, \kappa_k$ of configurations with $\kappa_j \vdash \kappa_{j+1}$, for all $0 \leq j < k$. We say that ρ ends in a state q if $q_k = q$, where q_k is the state in κ_k. We call ρ a ***run*** if $\kappa_0 = (q_I, \tau_I, \epsilon)$.

Example 3. In this example we will go a little way beyond our definition and consider a register pushdown *automaton* rather than a register pushdown *system*, so that we can motivate the definition of PDRS by looking at a particular

[2] Thus, register assignments here are *single* and *full* (*S F*).

[3] For technical reasons, it is convenient to have ε-transitions. However, to keep the definition minimal, we observe that they can be simulated with $push(1)$ followed by $pop(1)$.

language of words that is accepted. An r-PDRA is an r-PDRS equipped with an additional family of transitions $\{i\}_{i\in[r]}$ and a distinguished subset of final states. A transition i can be taken only if the next letter of the input word matches the contents of register i and the action of taking the transition is to consume the letter. The following example is a 2-PDRA accepting the language $\{ww^R \mid w \in \mathcal{D}^*\}$ of words w followed by their own reverse.



We draw the machine's only final state circled. The initial register assignment is unimportant, but let us say that it is $\{1 \mapsto a, 2 \mapsto b\}$. The behaviour of the machine is as follows. Starting from state q_0, the automaton first guesses a letter c from the infinite alphabet $\mathcal{D}$ to use as a 'bottom of stack' marker and stores it in register 2. To ensure that the guess is otherwise unconstrained, this is implemented by two consecutive $2^\bullet$ transitions (a single $2^\bullet$ transition would allow for guessing any letter of $\mathcal{D}$ under the constraint that it is different from a and b). The machine then pushes this 'bottom of stack' marker onto the stack and moves into the loop on state q_3. During this loop, the word w is consumed from the input by first guessing the next input letter using two consecutive $1^\bullet$ transitions and then reading it in using the 1 transition. The last part of the loop has the machine store each consumed letter on the stack. At the point at which the machine nondeterministically chooses to enter q_7 , the configuration has the form $(q_7, \{1 \mapsto d, 2 \mapsto c\}, wc)$ where d is the final letter of w. In the loop on q_7 the machine reads in w^R and, as in the finite-alphabet case, the reversal of the word is ensured by popping w off the stack. Finally, when the stack has been exhausted except for the 'bottom of stack' symbol c, then the machine may take the transition to q_2 and thus accept.

Remark 4. r-PDRS is meant to be a minimalistic model allowing us to study reachability in the infinite-alphabet setting with registers and pushdown storage. Existing related models [8, 22] feature transitions of a more compound shape, which can be readily translated into sequences of PDRS transitions.

For instance, a transition of an infinite-alphabet pushdown automaton [8] typically involves a refreshment ($i^\bullet$) followed by pop ($pop(j)$) and a sequence of pushes ($push(j)$). This decomposition leads to a linear blow-up in size for translations of reachability questions into the r-PDRS setting. For register pushdown automata [22], an additional complication is their use of *non-injective* register assignments. Observe, though, that transitions in the non-injective framework can be easily mimicked using injective register assignments provided we keep track of the partitions determined by duplicated values in the original automaton. The book-keeping can be implemented inside the control state, which leads to an exponential blow-up in the size of the system, because the number of all possible partitions is exponential. Note that the number of registers does not change during such a simulation. Another difference is that

register pushdown automata [22] are tailored towards data languages, i.e. a stack symbol is an element of $\mathcal{D}$ paired up with a tag drawn from a finite set. From this perspective, r-PDRSs use a singleton set of tags. Still, richer tag sets could be encoded via sequences of elements of $\mathcal{D}$ (for example, to simulate the i-th out of k tags, we could push sequences of the form $d_1^i d_2$ for $d_1, d_2 \in \mathcal{D}$ with $d_1 \neq d_2$). This reduction is achievable in polynomial time.

Following [13, 8, 20], we mostly use *injective* register assignments. This is done to allow us to explore whether the restriction still leads to asymptotically more efficient reachability testing, as in the pushdown-free case. On a foundational note, injectivity gives a more essential treatment of freshness with respect to a set of registers: non-injective assignments can easily be used to encode PSPACE computations that have little to do with the interaction between finite control (and pushdown) and freshness.

Name permutations. There is a natural action of the group of permutations of $\mathcal{D}$ on stacks, assignments, runs, etc. For instance, given a permutation $\pi : \mathcal{D} \to \mathcal{D}$ and an assignment τ, the result of applying π to τ is the register assignment $\pi \cdot \tau$ given by $\{(i, \pi(d)) \mid (i, d) \in \tau\}$. Similarly, $\pi \cdot s = \pi(d_n) \cdots \pi(d_1)$ for any stack $s = d_n \cdots d_1$ while, on the other hand, $\pi \cdot q = q$ for all states q. Hence, $\pi \cdot (q, \tau, s) = (q, \pi \cdot \tau, \pi \cdot s)$ and, for $\rho = \kappa_0 \vdash \cdots \vdash \kappa_n$ a transition sequence, $\pi \cdot \rho$ is the sequence $\pi \cdot \kappa_0, \cdots, \pi \cdot \kappa_n$.

Note that, as long as our constructions involve finitely many names, they will always have a finite support: we say that a set $S \subseteq \mathcal{D}$ *supports* some (nominal) element x if, for all permutations π, if $\pi(n) = n$ for all $n \in S$ then $\pi \cdot x = x$. Accordingly, ***the support*** $\nu(x)$ of x is the smallest set S supporting x. For example, $\nu(\tau) = \{\tau(i) \mid i \in [r]\}$, for all assignments τ. The support of a run $\rho = \kappa_0 \vdash \cdots \vdash \kappa_n$ is $\nu(\rho) = \bigcup_{j=0}^{n} \nu(\kappa_j)$, i.e. it consists of all elements of $\mathcal{D}$ that occur in it. The finite-support setting can be formally described by means of *nominal sets* [11] and closure results such as the following hold.

Fact 5 (Closure Under Permutations)**.** *Fix an r-PDRS and let ρ be a transition sequence and $\pi : \mathcal{D} \to \mathcal{D}$ a permutation. Then $\pi \cdot \rho$ is also a transition sequence.*

3. Distinguishability

Devices with r registers but without pushdown storage, such as finite-memory automata [13], can take advantage of the registers to distinguish r elements of $\mathcal{D}$ from the rest. Consequently, any run can be replaced with a run that ends in the same state, yet is supported by merely r elements of the infinite alphabet [13, Proposition 4].

With extra pushdown storage, an r-PDRS is capable of storing unboundedly many elements of $\mathcal{D}$. Nevertheless, the restricted nature of the stack makes it possible to place a finite bound on the size of the support needed for a run to a given state, which is again a function of the number of registers.

Lemma 6 (Limited Distinguishability)**.** *Fix an r-PDRS. For every transition sequence $\rho = (q_0, \tau_0, \epsilon) \vdash^n (q_n, \tau_n, \epsilon)$, there is a transition sequence $\rho' = (q_0, \tau'_0, \epsilon) \vdash^n (q_n, \tau'_n, \epsilon)$ with $\tau'_0 = \tau_0$, $\tau'_n = \tau_n$ and $|\nu(\rho')| \leq 3r$.*

Proof. The proof is by induction on n. When $n \leq 1$, the result is trivial. Otherwise, we distinguish two cases.

In the first case, the transition sequence is of the form:

$$(q_0, \tau_0, \epsilon) \vdash (q_1, \tau_1, d) \vdash^{n-2} (q_{n-1}, \tau_{n-1}, d) \vdash (q_n, \tau_n, \epsilon)$$

in which the first transition is by $push(i)$ (so $d = \tau_1(i)$), the last transition is by $pop(j)$ or $pop^\bullet$ and the stack does not empty until the final transition. Since d is never popped from the stack during the middle segment, also $(q_1, \tau_1, \epsilon) \vdash^{n-2} (q_{n-1}, \tau_{n-1}, \epsilon)$ is a valid transition sequence and hence, from the induction hypothesis, there is a transition sequence between the same two configurations using no more than $3r$ names. By adding d to the bottom of every stack in this sequence one obtains another valid transition sequence: $(q_1, \tau'_1, d) \vdash^{n-2} (q_{n-1}, \tau'_{n-1}, d)$ with $\tau'_1 = \tau_1$ and $\tau'_{n-1} = \tau_{n-1}$, and the new sequence features $\leq 3r$ names. It follows that the latter can be extended to the required:

$$(q_0, \tau_0, \epsilon) \vdash (q_1, \tau'_1, d) \vdash^{n-2} (q_{n-1}, \tau'_{n-1}, d) \vdash (q_n, \tau_n, \epsilon)$$

since neither $push(i)$, nor $pop(j)/pop^\bullet$ change the registers.

Otherwise, the transition sequence is of the form:

$$(q_0,\ \tau_0,\ \epsilon) \vdash^k (q_k,\ \tau_k,\ \epsilon) \vdash^{n-k} (q_n,\ \tau_n,\ \epsilon)$$

with $0 < k < n$. It follows from the induction hypothesis that there are sequences:

$$\begin{aligned} \rho_1 &= (q_0,\ \tau'_0,\ \epsilon) \vdash^k (q_k,\ \tau'_k,\ \epsilon) \\ \rho_2 &= (q_k,\ \tau'_k,\ \epsilon) \vdash^{n-k} (q_n,\ \tau'_n,\ \epsilon) \end{aligned}$$

with $\tau'_0 = \tau_0$, $\tau'_n = \tau_n$, $\tau'_k = \tau_k$ and which each, individually, use no more than $3r$ names. Let $N \supseteq \nu(\tau_0) \cup \nu(\tau_k) \cup \nu(\tau_n)$ be a set of names of size $3r$. We aim to map $\nu(\rho_1)$ and $\nu(\rho_2)$ into N by injections i and j respectively. For i we set $i(a) = a$ for any $a \in (\nu(\tau_0) \cup \nu(\tau_k))$ and otherwise choose some distinct $b \in N \setminus (\nu(\tau_0) \cup \nu(\tau_k))$. Similarly, for j we set $j(a) = a$ for any $a \in (\nu(\tau_k) \cup \nu(\tau_n))$ and otherwise choose some distinct $b \in N \setminus (\nu(\tau_k) \cup \nu(\tau_n))$. Note that these choices are always possible because $|\nu(\rho_1)| \leq |N| \geq |\nu(\rho_2)|$. Finally, we extend i and j to permutations π_i and π_j on $\mathcal{D}$. Since transition sequences are closed under permutations (Fact 5):

$$(q_0, \pi_i \cdot \tau_0, \epsilon) \vdash^k (q_k, \pi_i \cdot \tau_k = \pi_j \cdot \tau_k, \epsilon) \vdash^{n-k} (q_n, \pi_j \cdot \tau_n, \epsilon)$$

is a valid transition sequence with $\pi_i \cdot \tau_0 = \tau_0$, $\pi_j \cdot \tau_n = \tau_n$ and which is supported by a subset of N. □

Corollary 7. *Fix an r-PDRS $\mathcal{S}$ and a state q of $\mathcal{S}$. If there is a run of $\mathcal{S}$ ending in q then there is a run of $\mathcal{S}$ ending in q that is supported by at most $3r$ distinct names.*

The $3r$ bound given above is optimal in the sense that there exists an r-PDRS such that all runs to a certain state will have to rely on $3r$ elements of $\mathcal{D}$.

Lemma 8 (Most Discriminating r-PDRS)**.** *There exists an r-PDRS $\langle Q, q_I, \tau_I, \epsilon\rangle$ and $q \in Q$ such that $|\nu(\rho)| = 3r$ for any run ρ ending in q.*

Proof. Consider the following high-level description of an r-PDRS. The machine proceeds as follows:

1. Push registers in numerical order, twice, to obtain stack $\tau_I(r)\cdots\tau_I(1)\tau_I(r)\cdots\tau_I(1)$.
2. Refresh registers by performing $i^\bullet$ for all $1 \leq i \leq r$. Let the new assignment be τ_1.
3. Perform $pop^\bullet$ r-times, thus ensuring that, for each $1 \leq i, j \leq r$, $\tau_I(i) \neq \tau_1(j)$.
4. Push all registers in numerical order, to obtain stack $\tau_1(r)\cdots\tau_1(1)\tau_I(r)\cdots\tau_I(1)$.
5. Refresh all registers. Let the new assignment be τ_2.
6. Perform $pop^\bullet$ $2r$-times, thus ensuring that, for each i, j, $\tau_2(i) \neq \tau_1(j)$ and $\tau_2(i) \neq \tau_I(j)$.
7. Silently transition to state q.

Now observe that the conditions in steps 3 and 6 and the fact that register assignments are injective ensure that $|\nu(\tau_I) \cup \nu(\tau_1) \cup \nu(\tau_2)| = 3r$. Hence, any run reaching q is supported by exactly $3r$ distinct names. □

Remark 9. The $3r$ bound given above can be adapted to the automata presentations of [8, 22] yielding bounds $3r + \Theta(1)$. An adaptation of Lemma 8 improves upon Example 6 of [8], where a language requiring $2r-1$ different symbols was presented.

Being able to bound the number of registers is useful for obtaining reachability algorithms as it allows us to remove the complications of the infinite alphabet and reduce problems to the well-studied finite alphabet setting (e.g. Theorem 10).

4. Reachability is EXPTIME-complete

We consider the following decision problem.

r-PDRS Reach: Given an r-PDRS $\mathcal{S}$ and $q \in Q$, is there a run of $\mathcal{S}$ ending in q?

We shall show that the problem (and its counterparts for all the other closely related machine models) is EXPTIME-complete. Note that reachability is equivalent to language non-emptiness in the automata case.

4.1. Reachability is EXPTIME-solvable

Theorem 10. *r-PDRS Reach and language emptiness for infinite-alphabet pushdown automata [8] and register pushdown automata [22] are solvable in exponential time.*

Proof. Lemma 6 yields an exponential-time reduction of r-PDRS Reach to the classic reachability problem for pushdown systems over finite alphabets [5]: one can replace the r $\mathcal{D}$-valued registers with r $[3r]$-valued registers, and then incorporate them into the finite control (for a singly-exponential blow-up of the state space). Since the latter problem is solvable in polynomial time, it follows that r-PDRS Reach is in EXPTIME.

By Remark 4, the emptiness problem for infinite-alphabet pushdown automata [8] can be reduced to r-PDRS Reach in polynomial time, immediately yielding the EXPTIME upper bound[4]. For register pushdown automata [22] we have an exponential-time reduction to r-PDRS Reach, which does not yield the required bound. However, recall that the translation into r-PDRS preserves the number of registers, so Lemma 6 still implies a linear upper bound for the number of $\mathcal{D}$-values needed for finding an accepting run. Consequently, we can reduce language emptiness of register pushdown automata to a reachability problem for pushdown systems at an exponential cost. Since the latter is in P, the former is in EXPTIME. □

4.2. Reachability is EXPTIME-hard

The bound given above is tight: we simulate a polynomial-space Turing machine with a stack (i.e. a polynomial-space auxiliary pushdown automaton [9]), which has an EXPTIME-complete halting problem. A reduction from the more familiar alternating polynomial-space Turing machines would also be possible, but Cook's model is closer to r-PDRS, which allows us to concentrate on the main issue of encoding binary memory content without the need to model alternation.

Theorem 11. *r-PDRS Reach is EXPTIME-hard.*

We first give an outline of the argument before describing the proof in detail.

4.2.1. Argument outline

Let us assume an auxiliary pushdown automaton $\mathcal{M}$ working over a binary tape of size n and a stack alphabet of size k. We can assume WLOG that every transition of $\mathcal{M}$ performs a single push or pop (but not both) along with a read/write by the head followed by a head movement. Moreover, let us assume that the stack alphabet of $\mathcal{M}$ includes a distinguished symbol \$ marking the bottom of the stack and that, additionally, $\mathcal{M}$ starts with \$ on the stack and halts when it is popped (\$ is not used otherwise). Finally, we let the tape of $\mathcal{M}$ initially contain only 0's.

We shall construct a $(6n + k + 1)$-PDRS $\mathcal{S}$ such that $\mathcal{M}$ terminates iff $\mathcal{S}$ reaches a designated "final" state. The registers of $\mathcal{S}$ will be divided into 4 groups:

$$\tau = \boxed{\tau_{\langle 0\rangle}} \boxed{\tau_{\langle 1\rangle}} \boxed{\tau_{\langle 2\rangle}} \boxed{\tau_{\langle 3\rangle}}$$

of sizes k+1, $2n$, $2n$ and $2n$ respectively.

[4] Through a careful reading of the argument for emptiness in [8] one can infer an exponential upper bound, but here Lemma 6 gives a direct argument.

- $\tau_{\langle 0\rangle}$ will contain $\mathcal{D}$-values coding the k stack symbols and an auxiliary symbol # (we write $\hat{\#}$ for the letter which encodes #, etc.). It will be left untouched throughout the simulation.
- $\tau_{\langle 1\rangle}$ and $\tau_{\langle 2\rangle}$ will be used to encode the n-bit tape of $\mathcal{M}$ during push- and pop-transitions respectively.
- $\tau_{\langle 3\rangle}$ will have an auxiliary role and in particular will ensure the integrity of the tape when changing from pop- to push-transitions.

The simulation must maintain a representation of the current tape and stack content of $\mathcal{M}$ (as well as the current state, head position etc.). Since $\mathcal{S}$ is itself equipped with a pushdown stack, we shall use its stack to represent the stack of $\mathcal{M}$ by putting some distinguished k-element subset of $\mathcal{D}$ in bijection with the stack alphabet. A natural candidate for representing the tape content of $\mathcal{M}$ are the registers of $\mathcal{S}$. However, the requirement that the register assignment be an *injective* map whose range is $\mathcal{D}$ makes constructing an encoding difficult: in any given register assignment, there are no non-trivial relationships between the elements! Hence, to encode information we are forced to use the register assignment in conjunction with other data.

Let us consider two register assignments τ and τ' of length $2n$. We say that such a pair is *compatible* just if, for all $1 \leq j \leq n$:

$$\{\tau(2j-1), \tau(2j)\} \;=\; \{\tau'(2j-1), \tau'(2j)\}. \tag{1}$$

Although any given register assignment cannot encode any meaningful information, a pair of compatible register assignments can be used together to represent a tape t by a generalised xor, written $\tau[\tau']$:

$$\tau[\tau'](j) = 0 \;\text{ iff }\; \tau(2j) = \tau'(2j).$$

In other words, the tape content at position $1 \leq j \leq n$ is 0 if τ and τ' agree on the order of the data values $\{\tau(2j-1), \tau(2j)\}$ and 1 if they are the transposition of each other. We can think of the notation $\tau[\tau']$ as a binary function of τ and τ', in which case it has the property that any equation:

$$\tau[\tau'] = t$$

can be solved uniquely for τ, τ' or t (given the other parameters). This fact becomes essential to ensuring the integrity of the tape encoding when changing from simulating a pop to simulating a push transition. Note that although this binary function is commutative, the asymmetric notation reflects the fact that, by convention, we will view the first argument as being primary, often writing that τ is an encoding of t with respect to the *mask* τ'.

We construct $\mathcal{S}$ such that, when in a configuration with register assignment τ which is faithfully simulating a push transition of $\mathcal{M}$ with tape content t, we have $\tau_{\langle 1\rangle}[\tau_{I\langle 1\rangle}] = t$. When in such a configuration but simulating a pop transition, we have $\tau_{\langle 2\rangle}[\tau_{I\langle 2\rangle}] = t$. In other words, the masks used are just the initial register assignments to groups $\langle 1\rangle$ and $\langle 2\rangle$ respectively. Consequently, we have the invariant that, in any faithful simulation of $\mathcal{M}$ reaching a configuration with assignment τ, $\tau_{\langle 1\rangle}$ and $\tau_{I\langle 1\rangle}$ are compatible and $\tau_{\langle 2\rangle}$ and $\tau_{I\langle 2\rangle}$ are compatible.

At each step of the simulation, $\mathcal{S}$ needs to query its representation of the tape in order to determine which transition of $\mathcal{M}$ to apply, which entails computing, e.g. $\tau_{\langle 1\rangle}[\tau_{I\langle 1\rangle}]$. However, due to the injectivity constraint on register assignments, for $\mathcal{S}$ to be able to access a pair of compatible register assignments, at least one must be stored on the stack. Hence, $\mathcal{S}$ will use the stack not only to encode the stack content of $\mathcal{M}$, but also as part of the encoding of its tape content by storing copies of the masks $\tau_{I\langle 1\rangle}$ and $\tau_{I\langle 2\rangle}$ so that they are always accessible when needed. This dual use for the stack requires a considerable amount of bookkeeping.

The key problem to overcome is to ensure that the masks $\tau_{I\langle 1\rangle}$ and $\tau_{I\langle 2\rangle}$ are always available at the top of the stack when they are needed for decoding. Let us first consider $\tau_{I\langle 2\rangle}$ which is somewhat easier because it is used for pop transitions. Since $\mathcal{M}$ halts only with an empty stack, we know that for every pop transition made, there must have already been made a corresponding push transition. Hence, we arrange for $\mathcal{S}$, when simulating a push transition with a stack of height h, to additionally push an extra copy of $\tau_{I\langle 2\rangle}$ (which it has stored in its registers) so that it is available at the top of the stack when the machine next returns to a stack of height h, should it be required to simulate the corresponding pop transition.

Ensuring the availability of $\tau_{I\langle 1\rangle}$ for the purpose of simulating push transitions is much more difficult. It is not possible to "pre-load" the stack with copies of $\tau_{I\langle 1\rangle}$ in the same way as for $\tau_{I\langle 2\rangle}$ because (i) the length of sequences of consecutive push transitions is not known and (ii) the dual use of the stack requires that copies of $\tau_{I\langle 1\rangle}$ must be interleaved with data values simulating the letters of the stack alphabet pushed by $\mathcal{M}$. Matters are further complicated by the fact that looking up the value of $\tau_{I\langle 1\rangle}$ in order to compute $\tau_{\langle 1\rangle}[\tau_{I\langle 1\rangle}]$ requires popping it off the top of the stack. Consider the simulation of two consecutive push transitions. As part of the simulation of the first transition, $\tau_{I\langle 1\rangle}$ must be popped and hence lost. The injectivity constraint on the registers ensures that the only way that it could be preserved (by temporarily popping it into the registers) would be at the cost of losing $\tau_{\langle 1\rangle}$ instead. However, by the time the second transition is simulated, another copy is required to be accessible (i.e. be at the top of the stack again). We cannot "pre-load" the stack with copies of $\tau_{I\langle 1\rangle}$, we cannot temporarily store it in the registers and accessing a copy consumes it. Hence we are forced to construct $\mathcal{S}$ in such a way that it computes $\tau_{\langle 1\rangle}[\tau_{I\langle 1\rangle}]$ without accessing $\tau_{I\langle 1\rangle}$ at all. To achieve this, we arrange for $\mathcal{S}$ to *guess* a mask $\tau'_{\langle 1\rangle}$, compute $\tau_{\langle 1\rangle}[\tau'_{\langle 1\rangle}]$ with respect to this guess and place the guessed mask on the stack. During the later simulation of pop transitions, when $\mathcal{S}$ has some spare capacity in its registers because $\tau_{\langle 1\rangle}$ is no longer used to encode tape content, it will be able to discharge the obligation of verifying that it guessed $\tau'_{\langle 1\rangle} = \tau_{I\langle 1\rangle}$.

So far, we have discussed the need to store three kinds of entities on the stack, namely: representations of the stack letters of $\mathcal{M}$, copies of the mask $\tau_{I\langle 2\rangle}$ and unverified guesses of the mask $\tau_{I\langle 1\rangle}$. In fact, to be able to switch from tape encoding using register group $\langle 2\rangle$ to tape encoding using register group $\langle 1\rangle$ (which happens whenever $\mathcal{S}$ simulates $\mathcal{M}$ performing a pop followed by a push) without loss of information, we need to maintain additional entities.

More specifically, during its operation, the stack of $\mathcal{S}$ will have the form:

$$s = \begin{array}{|c|c|c|c|c|c|} \hline s^i_{\langle 0\rangle} & s^i_{\langle 1\rangle} & \tau_{I\langle 2\rangle} & s^i_{\langle 3\rangle} & \tau_{I\langle 2\rangle} & s^i_{\langle 3\rangle} \\ \hline s^{i-1}_{\langle 0\rangle} & s^{i-1}_{\langle 1\rangle} & \tau_{I\langle 2\rangle} & s^{i-1}_{\langle 3\rangle} & \tau_{I\langle 2\rangle} & s^{i-1}_{\langle 3\rangle} \\ \hline \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hline \end{array} \tag{2}$$

with $|s^i_{\langle 1\rangle}| = |\tau_{I\langle 2\rangle}| = |s^i_{\langle 3\rangle}| = 2n$ and $|s^i_{\langle 0\rangle}| = 1$ (and arbitrary i). The top stack symbol is in the top-left corner, the element below it on the stack is on its right, $s^{i-1}_{\langle 0\rangle}$ is below $s^i_{\langle 3\rangle}$ etc. Each $s^j_{\langle 0\rangle}$ above corresponds to a non-# stack symbol (as specified by $\tau_{\langle 0\rangle}$).

The guessed masks, to be verified later, will be stored as $s^j_{\langle 1\rangle}$. To represent the tape at pops, we will use the mask $\tau_{I\langle 2\rangle}$. Our simulation will make sure that this is correctly stored on the stack during pushes. Finally, the elements $s^j_{\langle 3\rangle}, \tau_{I\langle 2\rangle}, s^j_{\langle 3\rangle}$ will be used to support the simulation when it switches from pops to pushes. The element $s^j_{\langle 3\rangle}$ will also be a mask.

There will also be "exceptional" rows, which have the form:

$$s^i_{\#} = \begin{array}{|c|c|c|c|c|c|} \hline \hat{\#} & s^{i\prime}_{\langle 3\rangle} & \tau_{I\langle 2\rangle} & s^i_{\langle 1\rangle} & \tau_{I\langle 2\rangle} & s^i_{\langle 3\rangle} \\ \hline \end{array} \tag{3}$$

where $\hat{\#}$ is the encoding of # in $\tau_{I\langle 0\rangle}$. These rows will be used to verify the correctness of $s^j_{\langle 3\rangle}$, as we discuss later.

Thus, in each row we store a stack symbol and 5 masks. The purpose of the masks is to help us determine the tape content of $\mathcal{M}$. On the other hand, the current stack of $\mathcal{M}$ can be recovered by projecting out the first column of s and erasing any occurrence of $\hat{\#}$.

We next define the different steps of the simulation. Initially, there is an initialisation step pushing a row s^0 on the stack; $\mathcal{S}$ reaches its final state precisely when it successfully pops s^0. Recall that every transition of $\mathcal{M}$ includes a push or a pop action. Accordingly, the states of $\mathcal{S}$ have two *modes*: a push-mode and pop-mode. Each transition of $\mathcal{S}$ non-deterministically guesses the mode of the next state (of $\mathcal{S}$). $\mathcal{S}$ simulates the push-transitions of $\mathcal{M}$ using $\tau_{\langle 1\rangle}[s^i_{\langle 1\rangle}]$ as its tape; it simulates the pop ones using $\tau_{\langle 2\rangle}[\tau_{I\langle 2\rangle}]$ as its tape. The simulation only goes through if the masks of these encodings are correct, that is, if $s^i_{\langle 1\rangle} = \tau_{I\langle 1\rangle}$ for all i. Put otherwise, in an accepting computation of $\mathcal{S}$, all masks $s^i_{\langle 1\rangle}$ that appear on the stack must be equal to $\tau_{I\langle 1\rangle}$.

Initialisation. $\mathcal{S}$ starts off by pushing the following row:

$$s^0 = \begin{array}{|c|c|c|c|c|c|} \hline \hat{\$} & \tau_{I\langle 1\rangle} & \tau_{I\langle 2\rangle} & \tau_{I\langle 3\rangle} & \tau_{I\langle 2\rangle} & \tau_{I\langle 3\rangle} \\ \hline \end{array} \tag{4}$$

Recall here that the tape of $\mathcal{M}$ is assumed to be initially empty, and this is captured by our initial encoding: $\tau_{I\langle 1\rangle}[s^0_{\langle 1\rangle}] = 0 \cdots 0$ (since $\tau_{I\langle 1\rangle} = s^0_{\langle 1\rangle}$).

Push-mode. During this phase, $\mathcal{S}$ uses $\tau_{\langle 1\rangle}$ to represent the tape of $\mathcal{M}$, while $\tau_{\langle 0\rangle}, \tau_{\langle 2\rangle}, \tau_{\langle 3\rangle}$ stay unchanged. Assuming $\mathcal{S}$ has stack content as in (2) or (3) and the current register assignment is:

$$\tau = \begin{array}{|c|c|c|c|} \hline \tau_{I\langle 0\rangle} & \tau_{\langle 1\rangle} & \tau_{\langle 2\rangle} & \tau_{\langle 3\rangle} \\ \hline \end{array}$$

the current tape content (according to the simulation) is:

$$t = \tau_{\langle 1\rangle}[s^i_{\langle 1\rangle}]\,.$$

Moreover, by construction, we shall have $\tau_{\langle 2\rangle} = \tau_{I\langle 2\rangle}$.

Now suppose $\mathcal{M}$ is to perform a push-transition $q \xrightarrow{x,y,z,push(C)} q'$, where x is the currently scanned symbol to be overwritten with y, the head movement is indicated by $z \in \{L, R, N\}$ and C is the stack symbol to be pushed. Assume that the current head position is $1 \leq j \leq n$ ($\mathcal{S}$ will keep track of it in its state). In order to simulate the transition, $\mathcal{S}$ first needs to retrieve $\tau_{\langle 1\rangle}[s^i_{\langle 1\rangle}](j)$. One way to do that would be to simply pop $s^i_{\langle 1\rangle}$ off the stack. However, that would be catastrophic for our simulation since there is no guarantee at this point that $s^i_{\langle 1\rangle}$ is a correct mask (i.e. equal to $\tau_{I\langle 1\rangle}$); popping it would destroy it, thus annihilating any possibility of verifying its correctness.[5] Thus, instead, $\mathcal{S}$ will *guess* what the value of $s^i_{\langle 1\rangle}$ might be and operate according to the guess. Moreover, the guess will be pushed on the stack for subsequent verification in the pop-mode.

More precisely, $\mathcal{S}$ first pushes the word $\tau_{\langle 2\rangle}\tau_{\langle 3\rangle}\tau_{\langle 2\rangle}\tau_{\langle 3\rangle}$. It then produces a guess $s^{i+1}_{\langle 1\rangle}$ of the mask $s^i_{\langle 1\rangle}$ which is consistent with reading x at the current head position, in that $\tau_{\langle 1\rangle}[s^{i+1}_{\langle 1\rangle}](j) = x$. This is achieved by non-deterministically pushing consecutive pairs of elements of $\tau_{\langle 1\rangle}$, subject to the previous requirement and the constraint expressed by equation (1). It next performs the operations of $\mathcal{M}$ (as instructed by y and z) according to the tape $\tau_{\langle 1\rangle}[s^{i+1}_{\langle 1\rangle}]$. Thus, after these operations, the register assignment and top of the stack read

$$\tau' = \boxed{\tau_{I\langle 0\rangle}}\boxed{\tau'_{\langle 1\rangle}}\boxed{\tau_{I\langle 2\rangle}}\boxed{\tau_{\langle 3\rangle}} \qquad s^{i+1} = \boxed{}\boxed{s^{i+1}_{\langle 1\rangle}}\boxed{\tau_{I\langle 2\rangle}}\boxed{\tau_{\langle 3\rangle}}\boxed{\tau_{I\langle 2\rangle}}\boxed{\tau_{\langle 3\rangle}}$$

where $\tau'_{\langle 1\rangle}[s^{i+1}_{\langle 1\rangle}]$ is the new tape content (in particular, we have written y in bit j).

Finally, $\mathcal{S}$ chooses the kind of the next transition. If it is a push, then $\hat{C}$ is pushed on the stack (the name corresponding to C). Otherwise, before switching to pop-mode, it has to transfer its tape-encoding from $\tau_{\langle 1\rangle}$ to $\tau_{\langle 2\rangle}$. We achieve this by simultaneously moving $s^{i+1}_{\langle 1\rangle}$ from the stack into $\tau'_{\langle 1\rangle}$ and changing $\tau_{I\langle 2\rangle}$ into $\tau'_{\langle 2\rangle}$ so that $\tau'_{\langle 2\rangle}[\tau_{I\langle 2\rangle}] = \tau'_{\langle 1\rangle}[s^{i+1}_{\langle 1\rangle}]$. This can be executed by popping $s^{i+1}_{\langle 1\rangle}(2j)$ and comparing it with $\tau'_{\langle 1\rangle}(2j)$ for each bit $1 \leq j \leq n$ and, then, depending on the outcome, swapping the contents of registers $2j-1$ and $2j$ in each of $\tau'_{\langle 1\rangle}$ and $\tau_{\langle 2\rangle}$ in order to reflect the desired change. Finally, we push $s^{i+1}_{\langle 1\rangle}$ back on the stack, followed by $\hat{C}$ to arrive at

$$\tau' = \boxed{\tau_{I\langle 0\rangle}}\boxed{s^{i+1}_{\langle 1\rangle}}\boxed{\tau'_{\langle 2\rangle}}\boxed{\tau_{\langle 3\rangle}} \qquad s^{i+1} = \boxed{\hat{C}}\boxed{s^{i+1}_{\langle 1\rangle}}\boxed{\tau_{I\langle 2\rangle}}\boxed{\tau_{\langle 3\rangle}}\boxed{\tau_{I\langle 2\rangle}}\boxed{\tau_{\langle 3\rangle}}$$

with $\tau'_{\langle 2\rangle}[\tau_{I\langle 2\rangle}] = \tau'_{\langle 1\rangle}[s^{i+1}_{\langle 1\rangle}]$.

Pop-mode. $\mathcal{S}$ now uses $\tau_{\langle 2\rangle}$ as the tape. Let the current configuration of $\mathcal{S}$ have stack as in (2) and let

$$\tau = \boxed{\tau_{I\langle 0\rangle}}\boxed{\tau_{\langle 1\rangle}}\boxed{\tau_{\langle 2\rangle}}\boxed{\tau_{\langle 3\rangle}}$$

[5] Note that pushing initially $\tau_{I\langle 1\rangle}$ an indefinite amount of times is not a viable solution for having it available on the stack as a mask, since the push operations of $\mathcal{M}$ would just bury these masks in the stack.

so that $\tau_{\langle 2\rangle}[\tau_{I\langle 2\rangle}]$ is the represented tape content.

Suppose that $\mathcal{M}$'s head scans position j and the next transition is to be $q \xrightarrow{x,y,z,pop(C)} q'$, where $s^i_{\langle 0\rangle} = \hat{C}$. Recall that $\tau_{\langle 1\rangle}, s^i_{\langle 1\rangle}, \cdots, s^1_{\langle 1\rangle}$ are guesses from previous push transitions (or from exceptional pushes, still to be discussed), while $s^0_{\langle 1\rangle} = \tau_{I\langle 1\rangle}$ by (4). To verify the guesses, it suffices to verify that $\tau_{\langle 1\rangle} = s^i_{\langle 1\rangle}$ and, at the next pop, verify that $\tau_{\langle 1\rangle} = s^{i-1}_{\langle 1\rangle}$, etc. Thus, $\mathcal{S}$ will first pop $\hat{C}$ from the stack and then pop $s^i_{\langle 1\rangle}$, simultaneously checking that it equals $\tau_{\langle 1\rangle}$ (otherwise it will block). Next we pop $\tau_{I\langle 2\rangle}$ to determine $\tau_{\langle 2\rangle}[\tau_{I\langle 2\rangle}]$ and perform the instruction (x, y, z) by changing $\tau_{\langle 2\rangle}$ to $\tau'_{\langle 2\rangle}$ (and updating the head position in the state). Thus, we obtain

$$\tau' = \boxed{\tau_{I\langle 0\rangle} \mid \tau_{\langle 1\rangle} \mid \tau'_{\langle 2\rangle} \mid \tau_{\langle 3\rangle}} \qquad s^{i\prime} = \boxed{ \mid \mid \mid s^i_{\langle 3\rangle} \mid \tau_{I\langle 2\rangle} \mid s^i_{\langle 3\rangle}}$$

having verified that $\tau_{\langle 1\rangle} = s^i_{\langle 1\rangle}$.

Now $\mathcal{S}$ guesses the kind of the next transition. If it is to be a pop, $\mathcal{S}$ pops the remainder of $s^{i\prime}$. Otherwise, $\mathcal{S}$ should switch to push-mode and in particular change its tape-storing routine from $\tau_{\langle 2\rangle}$ to $\tau_{\langle 1\rangle}$. That is, $\tau_{\langle 1\rangle}$ needs to be updated to $\tau'_{\langle 1\rangle}$ such that $\tau'_{\langle 1\rangle}[\hat{\tau}_{\langle 1\rangle}] = t$, for an appropriate mask $\hat{\tau}_{\langle 1\rangle}$, where $t = \tau'_{\langle 2\rangle}[\tau_{I\langle 2\rangle}]$ is the current tape. Also, having now preserved its value according to the encoding, $\tau'_{\langle 2\rangle}$ should be changed back to $\tau_{I\langle 2\rangle}$.

We are now faced with the following obstacle. Updating $\tau_{\langle 1\rangle}$ to $\tau'_{\langle 1\rangle}$ would make us lose the current $\tau_{\langle 1\rangle}$. This would break our simulation as, although $\mathcal{S}$ has verified $\tau_{\langle 1\rangle} = s^i_{\langle 1\rangle}$, it remains to check that $\tau_{\langle 1\rangle}$ is the same as all those guessed masks still on the stack, i.e. $s^{i-1}_{\langle 1\rangle}, \ldots, s^1_{\langle 1\rangle}$. Since entering push-mode requires that we overwrite the contents of $\tau_{\langle 1\rangle}$, we would like to preserve its current assignment on the stack. However, doing this directly is impossible since we need to obtain $\tau_{I\langle 2\rangle}$ from lower down the stack in order to decode the current tape contents $\tau'_{\langle 2\rangle}$. We overcome this by storing in the stack a guess for $\tau_{\langle 1\rangle}$, along with auxiliary masks which will allow us to verify this guess when popping. Moreover, we shall pick $\hat{\tau}_{\langle 1\rangle} = \tau_{\langle 1\rangle}$. Hence, $\mathcal{S}$ operates as follows.

- It first pops $s^i_{\langle 3\rangle}$ and stores it in $\tau_{\langle 3\rangle}$.
- Then, it pops $\tau_{I\langle 2\rangle}$ and stores it in $\tau'_{\langle 2\rangle}$ and, at the same time, it copies t in $\tau_{\langle 1\rangle}$ and $\tau_{\langle 3\rangle}$, that is, it updates them to $\tau'_{\langle 1\rangle}$ and $\tau'_{\langle 3\rangle}$ respectively, such that $t = \tau'_{\langle 1\rangle}[\tau_{\langle 1\rangle}] = \tau'_{\langle 3\rangle}[s^i_{\langle 3\rangle}]$.
- It pushes $\tau_{I\langle 2\rangle}$ back on the stack.
- It then makes a guess $\tau''_{\langle 1\rangle}$ of $\tau_{\langle 1\rangle}$ and pushes it on the stack. While doing so, it updates $\tau'_{\langle 3\rangle}$ to the mask $s^{i\prime}_{\langle 3\rangle}$ satisfying $\tau'_{\langle 1\rangle}[\tau''_{\langle 1\rangle}] = \tau'_{\langle 3\rangle}[s^{i\prime}_{\langle 3\rangle}]$.
- Finally, it pushes $\hat{\#} s^{i\prime}_{\langle 3\rangle} \tau_{I\langle 2\rangle}$ on the stack.

Thus, we end up with

$$\tau' = \boxed{\tau_{I\langle 0\rangle} \mid \tau'_{\langle 1\rangle} \mid \tau_{I\langle 2\rangle} \mid s^{i\prime}_{\langle 3\rangle}} \qquad s^i_{\#} = \boxed{\hat{\#} \mid s^{i\prime}_{\langle 3\rangle} \mid \tau_{I\langle 2\rangle} \mid \tau''_{\langle 1\rangle} \mid \tau_{I\langle 2\rangle} \mid s^i_{\langle 3\rangle}}$$

and the automaton switches to push-mode. Note that $\tau'_{\langle 1\rangle}$ uses $\tau_{\langle 1\rangle}$ as a mask, yet we have $\tau''_{\langle 1\rangle}$ on the stack instead. Nevertheless, by construction:

$$\begin{aligned} & \tau''_{\langle 1\rangle} = \tau_{\langle 1\rangle} \\ \text{iff} \quad & \tau'_{\langle 1\rangle}[\tau''_{\langle 1\rangle}] = \tau'_{\langle 1\rangle}[\tau_{\langle 1\rangle}] \\ \text{iff} \quad & \tau'_{\langle 3\rangle}[s^i_{\langle 3\rangle}] = \tau'_{\langle 3\rangle}[s^{i\prime}_{\langle 3\rangle}] \\ \text{iff} \quad & s^i_{\langle 3\rangle} = s^{i\prime}_{\langle 3\rangle}. \end{aligned}$$

Thus, when popping the row $s^i_{\#}$, the automaton will additionally check that $s^{i\prime}_{\langle 3\rangle} = s^i_{\langle 3\rangle}$ and thus verify that the correct mask $\tau''_{\langle 1\rangle}$ has been stored on the stack.

Finally, we discuss the case when $\mathcal{S}$ is in pop-mode and the top of the stack is as in (3). In such a case, the pop transition of $\mathcal{S}$ is taken independently of $\mathcal{M}$. In particular, let $\tau =$ $\boxed{\tau_{I\langle 0\rangle}}\boxed{\tau_{\langle 1\rangle}}\boxed{\tau_{\langle 2\rangle}}\boxed{\tau_{\langle 3\rangle}}$.

- $\mathcal{S}$ starts by popping $\hat{\#}$ and $s^{i\prime}_{\langle 3\rangle}$, and stores the latter in $\tau_{\langle 3\rangle}$.
- Next, it pops $\tau_{I\langle 2\rangle}$ and $s^i_{\langle 1\rangle}$, and checks that the latter is equal to $\tau_{\langle 1\rangle}$.
- Finally, it pops $\tau_{I\langle 2\rangle}$ and $s^i_{\langle 3\rangle}$, and checks that the latter is equal to $s^{i\prime}_{\langle 3\rangle}$ (which is now stored in $\tau_{\langle 3\rangle}$).

Thus, according to our discussion above, $\mathcal{S}$ correctly verifies the continuity of the masks $s^i_{\langle 1\rangle}$ while consuming the row $s^i_{\#}$.

Altogether, the above construction yields a $(6n + k + 1)$-PDRS $\mathcal{S}$ of polynomial size with respect to n. Moreover, $\mathcal{S}$ pops s^0 iff it makes consistent guesses for masks used in its first component and, therefore, faithfully simulates the operations of $\mathcal{M}$ leading to popping the terminating symbol from its stack.

4.2.2. Argument in detail

We now make the above sketch more precise.

Proof. Let $\mathcal{M} = \langle Q, q_I, T, \delta\rangle$ be an auxiliary pushdown automaton operating over a binary tape of size n and stack alphabet $\{1, \cdots, k\}$, with $\$ \in \{1, \cdots, k\}$ being a distinguished bottom-of-stack symbol. $\mathcal{M}$ has initial state q_I, and let us assume its transition relation is of the following type, $\delta \subseteq Q \times Op'_k \times Q$, with

$$Op'_k = \{0, 1\}^2 \times \{L, R, N\} \times \{push(i), pop(i) \mid 1 \leq i \leq k\}.$$

That is, in every transition, $\mathcal{M}$ reads a bit x from its current head position, writes back y, moves the head (Left, Right, or No-move), and pushes or pops a symbol from the stack. Moreover, we assume that $\mathcal{M}$ has initial tape $0 \cdots 0$ and initial stack \$, and $\mathcal{M}$ halts when it pops \$ from the stack (\$ is not used otherwise).

We construct a $(k + 1 + 6n)$-PDRS $\mathcal{S}$ such that $\mathcal{M}$ pops \$ from the stack iff $\mathcal{S}$ reaches a designated state q'_F. In particular,

$$\mathcal{S} = \langle (Q \times \{1, \cdots, n\} \times \{\uparrow, \downarrow\}) \cup \{q_I\} \cup Q', q_I, \tau_I, \delta'\rangle$$

where each state of the form $(q, j, \downarrow)$ [resp. $(q, j, \uparrow)$] is said to be in *push-mode* [*pop-mode*]. The index j indicates the position of the head on the tape of $\mathcal{M}$. Thus, at its initial position, $\mathcal{M}$ reads the first bit of the tape and can only perform a push. Q' is a set of auxiliary states of polynomial size in n, which we gradually specify below. The initial assignment τ_I is arbitrary; we divide the registers into 4 groups (of sizes $k+1$, $2n$, $2n$ and $2n$ respectively) and write register assignments τ in the form:

$$\tau \;=\; \tau_{\langle 0\rangle} :: \tau_{\langle 1\rangle} :: \tau_{\langle 2\rangle} :: \tau_{\langle 3\rangle}.$$

We moreover stipulate that, throughout the operation of $\mathcal{S}$, the values of its group-0 registers remain fixed and, for each $i = 1, 2, 3$ and $1 \leq j \leq n$,

$$\{\tau_{\langle i\rangle}(2j-1), \tau_{\langle i\rangle}(2j)\} \;=\; \{\tau_{I\langle i\rangle}(2j-1), \tau_{I\langle i\rangle}(2j)\}\,.$$

The discipline imposed above is instrumented so as to encode an n-size tape in each of $\tau_{\langle 1\rangle}, \tau_{\langle 2\rangle}, \tau_{\langle 3\rangle}$. In particular, for each $i = 1, 2, 3$ and $2n$-components $\tau_{\langle i\rangle}, \hat{\tau}_{\langle i\rangle}$, we can define an n-tape $\tau_{\langle i\rangle}[\hat{\tau}_{\langle i\rangle}]$ by setting,

$$\tau_{\langle i\rangle}[\hat{\tau}_{\langle i\rangle}](j) = 0 \;\text{ iff }\; \tau_{\langle i\rangle}(2j) = \hat{\tau}_{\langle i\rangle}(2j)$$

for all $1 \leq j \leq n$. We call $\hat{\tau}_{\langle i\rangle}$ *the mask* of the encoding. Our PDRS $\mathcal{S}$ will simulate the operations of $\mathcal{M}$ using just such an encoding. On the other hand, the stack is grouped into rows of two possible forms:

$$s^i \;=\; s^i_{\langle 0\rangle} :: s^i_{\langle 1\rangle} :: s^i_{\langle 2\rangle} :: s^i_{\langle 3\rangle} :: s^i_{\langle 2\rangle} :: s^i_{\langle 3\rangle} \tag{5}$$

$$s^i_{\#} \;=\; \hat{\#} :: s^{i\prime}_{\langle 3\rangle} :: s^i_{\langle 2\rangle} :: s^i_{\langle 1\rangle} :: s^i_{\langle 2\rangle} :: s^i_{\langle 3\rangle} \tag{6}$$

where the first component has size 1, and the rest have size $2n$. In rows of the form (5), we stipulate that $s^i_{\langle 0\rangle} \neq \hat{\#}$. Here $\hat{\#} = \tau_{I\langle 0\rangle}(k+1)$ is a symbol we use precisely for distinguishing stack rows like (6), which we call *exceptional* rows. Note that components with index $\langle 2\rangle$ and $\langle 3\rangle$ are repeated in (5); while components with index $\langle 2\rangle$ are repeated in (6). The index i in $s^i, s^i_{\#}$ denotes the i-th row of the stack. The i-th row is on top of the $(i-1)$-th, etc. At the bottom of the stack we will store the row:

$$s^0 \;=\; \tau_{I\langle 0\rangle}(\$) :: \tau_{I\langle 1\rangle} :: \tau_{I\langle 2\rangle} :: \tau_{I\langle 3\rangle} :: \tau_{I\langle 2\rangle} :: \tau_{I\langle 3\rangle} \tag{7}$$

where recall that $\$ \in \{1, \cdots, k\}$ is the empty stack symbol.

We proceed to define δ'. As explained in the main text, we divide the operations of $\mathcal{S}$ into operations performed in push-mode and pop-mode. In push mode, the automaton uses the $\langle 1\rangle$-component of its registers for encoding the tape, using a mask which it needs to push on the stack at every step. As different push-steps do not share their masks (the registers store the encoded tape, not the mask used), these may in general differ. $\mathcal{S}$ correctly simulates the operations of $\mathcal{M}$ if the same mask is used in every push-step. Thus, $\mathcal{S}$ stores on the stack the mask used for each such step and, when in pop-mode, it verifies the consistency of those chosen masks. In particular, we impose that in every correct computation of $\mathcal{S}$, all masks used in push-mode coincide with $\tau_{I\langle 1\rangle}$. In pop-mode, the automaton verifies the

consistency of the push-mode masks and uses instead the $\langle 2\rangle$-component of the registers for storing the tape. The mask used for that encoding is $\tau_{I\langle 2\rangle}$, which is guaranteed to be readily available on the stack as, by construction every $s^i_{\langle 2\rangle}$ appearing on the stack (in rows (5) or (6)) will satisfy $s^i_{\langle 2\rangle} = \tau_{I\langle 2\rangle}$.

We next proceed to the formal definition of the transitions of $\mathcal{S}$. We define δ' as the least relation containing the following transitions.[6]

Push-mode. The automaton starts its operation by storing s^0 on the stack. That is, we include in δ' the transition sequences

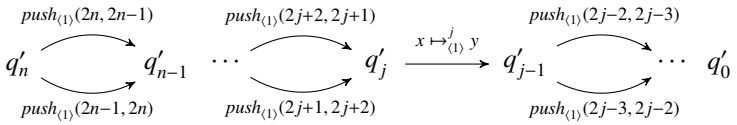
$$q_I \xrightarrow{push_{\langle 1,2,3,2,3\rangle}} \xrightarrow{push_{\langle 0\rangle}(\$)} (q_I, 0, \downarrow),$$
$$q_I \xrightarrow{push_{\langle 1,2,3,2,3\rangle}} \xrightarrow{push_{\langle 0\rangle}(\$)} (q_I, 0, \uparrow),$$

which simply push on the stack the contents of registers $\{\$\} \cup \{k+2, \cdots, 6n+k+1\}$ in the required fashion and guess whether the next state is going to be in push- or pop-mode.

We next move to ordinary push transitions. For each $(q, x, y, z, push(i), q') \in \delta$ and $j \in \{1, \cdots, n\}$, we include the following transition sequences.

$$(q, j, \downarrow) \xrightarrow{push_{\langle 2,3,2,3\rangle}} \xrightarrow{guess_{\langle 1\rangle}(q', j, x, y, z)} (q', j+z, i, \downarrow)'$$
$$(q', j+z, i, \downarrow)' \xrightarrow{push_{\langle 0\rangle}(i)} (q', j+z, \downarrow)$$
$$(q', j+z, i, \downarrow)' \xrightarrow{switch(q', j+z, i, \uparrow)} (q', j+z, i, \uparrow)' \xrightarrow{push_{\langle 0\rangle}(i)} (q', j+z, \uparrow)$$

with $j + z = j - 1$ if $z = L$, etc. All primed states are taken from Q'. Moreover, $guess_{\langle 1\rangle}(q', j, x, y, z)$ is the gadget:



with each q'_i being some $(q', j, x, y, z, i, \downarrow)'_g \in Q'$, for $i = 1, \cdots, n$, and $q'_0 = (q', j+z, i, \downarrow)'$. The transition sequence $x \mapsto^j_{\langle 1\rangle} y$ assumes that the j-th bit of the tape (as encoded in the $\langle 1\rangle$-component) is x, pushes the corresponding mask registers on the stack (according to the value of x), and changes the value of the j-th bit to y:

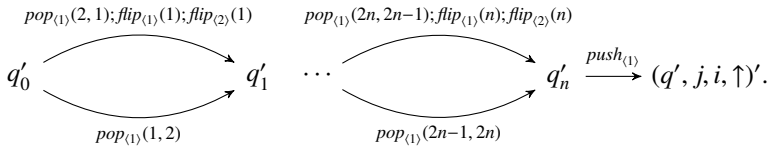
$$x \mapsto^j_{\langle 1\rangle} y = \begin{cases} push_{\langle 1\rangle}(2j-1, 2j) & \text{if } x = 0, y = 0 \\ push_{\langle 1\rangle}(2j-1, 2j); flip_{\langle 1\rangle}(j) & \text{if } x = 0, y = 1 \\ push_{\langle 1\rangle}(2j, 2j-1) & \text{if } x = 1, y = 1 \\ push_{\langle 1\rangle}(2j, 2j-1); flip_{\langle 1\rangle}(j) & \text{if } x = 1, y = 0 \end{cases}.$$

[6]Some notation: we write $push_{\langle 0\rangle}(i)$ for $push(i)$, $push_{\langle 1\rangle}(i)$ for $push(k{+}1{+}i)$, etc. Similarly for pops, and for referring to registers in general: e.g. $j^\bullet_{\langle 2\rangle} = (k{+}1{+}2n{+}j)^\bullet$. We write $q \xrightarrow{t_1;t_2} q'$ for the sequence of transitions $q \xrightarrow{t_1} q'' \xrightarrow{t_2} q'$, choosing some unique $q'' \in Q'$. Finally, we push sequences from right to left, i.e. write $push_{\langle i\rangle}(i_1, \cdots, i_m)$ for $push_{\langle i\rangle}(i_m); \cdots; push_{\langle i\rangle}(i_1)$, and let $push_{\langle i\rangle}$ abbreviate $push_{\langle i\rangle}(1, \cdots, 2n)$ (push the full i-th component), while e.g. $push_{\langle 1,2\rangle}$ stands for $push_{\langle 2\rangle}; push_{\langle 1\rangle}$. Dually for pops (in particular, we pop sequences left-to-right).

The transition sequence $\mathit{flip}_{\langle i\rangle}(j)$ $(i = 1, 2)$ simply flips the j-th bit of the tape in the $\langle i\rangle$-component of the registers:

$$\mathit{flip}_{\langle i\rangle}(j) \;=\; \mathit{push}_{\langle i\rangle}(2j{-}1, 2j); (2j)^{\bullet}_{\langle i\rangle}; (2j{-}1)^{\bullet}_{\langle i\rangle}; (2j)^{\bullet}_{\langle i\rangle}; \mathit{pop}_{\langle i\rangle}(2j, 2j{-}1).$$

Thus, the transition sequence starts by performing a push of the $\langle 2\rangle\langle 3\rangle\langle 2\rangle\langle 3\rangle$-components on the stack. It then guesses a mask for $\tau_{\langle 1\rangle}$, say s^{i+1}, and pushes it on the stack, while at the same time it updates $\tau_{\langle 1\rangle}$ to $\tau'_{\langle 1\rangle}$ according to the write instruction y. At that point, it guesses whether it needs to switch to pop-mode. If it guesses that it should remain in push-mode, then it simply pushes the stack symbol i (i.e. its representation in $\tau_{I\langle 0\rangle}$). Otherwise, it performs a switch by using $\mathit{switch}(q', j, i, \uparrow)$, which is the following gadget.



Here each q'_i is some $(q', j, i, \uparrow)'_s \in Q'$ $(i = 1, \cdots, n)$ and $q'_0 = (q', j{+}z, i, \downarrow)'$. The above allows us to switch our tape-storing routine from the $\langle 1\rangle$-component of the registers to the $\langle 2\rangle$-component: $\mathcal{S}$ pops the mask $s^{i+1}_{\langle 1\rangle}$ from the stack and stores it in the $\langle 1\rangle$-component of the registers and, while doing so, it copies the value of $\tau'_{\langle 1\rangle}[s^{i+1}]$ onto $\tau_{\langle 2\rangle}$. This is achieved via the $\mathit{flip}_{\langle 2\rangle}(i)$ transitions, in case $\tau'_{\langle 1\rangle}[s^{i+1}] = 1$ (upper arcs); and by their alternatives, which leave $\tau_{\langle 2\rangle}(2i)$ and $\tau_{\langle 2\rangle}(2i - 1)$ untouched, if $\tau'_{\langle 1\rangle}[s^{i+1}] = 0$ (lower arcs).

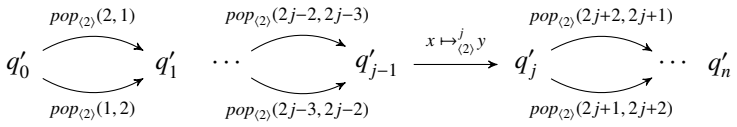
Pop-mode. For each $(q, x, y, z, \mathit{pop}(i), q') \in \delta$ and $j \in \{1, \cdots, n\}$, include the following transition sequences,

$$(q, j, \uparrow) \xrightarrow{\mathit{pop}_{\langle 0\rangle}(i)} \xrightarrow{\mathit{pop}_{\langle 1\rangle}} \xrightarrow{\mathit{read}_{\langle 2\rangle}(q', j, x, y, z)} (q', j + z, \uparrow)'$$

$$(q', j + z, \uparrow)' \xrightarrow{\mathit{clean}_{\langle 3\rangle}} \xrightarrow{\mathit{clean}_{\langle 2\rangle}} \xrightarrow{\mathit{clean}_{\langle 3\rangle}} (q', j + z, \uparrow)$$

$$(q', j{+}z, \uparrow)' \xrightarrow{\mathit{switch}(q', j+z, \downarrow)} (q', j{+}z, \downarrow)' \xrightarrow{\mathit{push}_{\langle 0\rangle}(k+1)} (q', j{+}z, \downarrow)$$

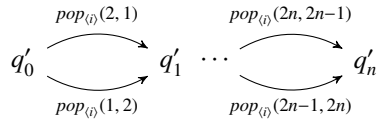
with $j + z$ as above and all primed states taken from Q'. Note that, be definition, $\mathit{pop}_{\langle 1\rangle}$ pops the $\langle 1\rangle$-component off the top of the stack and checks that it is the same as the $\langle 1\rangle$-component of the registers. On the other hand, $\mathit{read}_{\langle 2\rangle}(q', j, x, y, z)$ is the gadget:



with $q'_0, q'_1, \cdots, q'_n \in Q'$.[7] The transition sequence $x \mapsto^j_{\langle 2\rangle} y$ is given by:

$$x \mapsto^j_{\langle 2\rangle} y = \begin{cases} pop_{\langle 2\rangle}(2j{-}1, 2j) & \text{if } x = 0, y = 0 \\ pop_{\langle 2\rangle}(2j{-}1, 2j); flip_{\langle 2\rangle}(j) & \text{if } x = 0, y = 1 \\ pop_{\langle 2\rangle}(2j, 2j{-}1) & \text{if } x = 1, y = 1 \\ pop_{\langle 2\rangle}(2j, 2j{-}1); flip_{\langle 2\rangle}(j) & \text{if } x = 1, y = 0 \end{cases}.$$

Thus, the transition sequence starts by popping the stack symbol i and checking that the last guessed mask for $\tau_{\langle 1\rangle}$ stored in registers equals the one stored at the top of the stack. It then continues to pop the mask $\tau_{I\langle 2\rangle}$ for $\tau_{\langle 2\rangle}$, focussing specifically in the values determining the value of its j-th bit. It verifies that $\tau_{\langle 2\rangle}[\tau_{I\langle 2\rangle}](j) = x$ and updates the latter to y (yielding some $\tau'_{\langle 2\rangle}$). At this point, it guesses whether the next state is going to be in pop-mode. If it guesses so, it simply pops the remaining top-row from the stack (without looking at the popped values). The latter is accomplished by the $clear_{\langle i\rangle}$ gadgets:



for appropriately chosen $q'_0, \cdots, q'_n \in Q'$. We specifically stipulate that if $i = \$$ (the symbol signifying termination) then $\mathcal{S}$ will necessarily guess to stay in pop-mode and, instead of reaching state $(q', j + z, \uparrow)$, it will drive itself to a designated state $q'_F \in Q'$.

If, on the other hand, the automaton guesses that the next state is in push-mode, it makes a switch of modes, using the $switch(q', j, \downarrow)$ gadget, and pushes the special symbol $\tau_{I\langle 0\rangle}(k + 1)$ on the stack. The $switch(q', j, \downarrow)$ gadget is decomposed as follows:

$$(q', j{+}z, \uparrow)' \xrightarrow{pop\text{-}in_{\langle 3\rangle}} \xrightarrow{pop\text{-}in_{\langle 2\rangle}\&copy\text{-}in_{\langle 1,3\rangle}} \xrightarrow{push_{\langle 2\rangle}} \xrightarrow{guess_{\langle 1\rangle}\&copy\text{-}in_{\langle 3\rangle}} \xrightarrow{push_{\langle 3,2\rangle}} (q', j{+}z, \downarrow)'.$$

Recall that at this point in the simulation, the register assignment and top of the stack of $\mathcal{S}$ are as follows:

$$\tau' = \tau_{I\langle 0\rangle} :: \tau_{\langle 1\rangle} :: \tau'_{\langle 2\rangle} :: \tau_{\langle 3\rangle} \quad s^{i\prime} = s^i_{\langle 3\rangle} :: \tau_{I\langle 2\rangle} :: s^i{}_{\langle 3\rangle}.$$

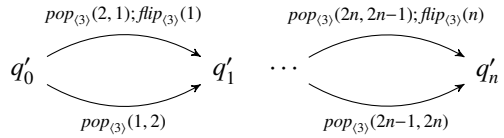
As described in the main text, the utility of the above is to:

- First pop $s^i_{\langle 3\rangle}$ and store it in $\tau_{\langle 3\rangle}$ (via $pop\text{-}in_{\langle 3\rangle}$).
- Then, pop $\tau_{I\langle 2\rangle}$ and store it in $\tau'_{\langle 2\rangle}$ and, at the same time, copy $t = \tau'_{\langle 2\rangle}[\tau_{I\langle 2\rangle}]$ in $\tau_{\langle 1\rangle}$ and $\tau_{\langle 3\rangle}$, that is, update them to $\tau'_{\langle 1\rangle}$ and $\tau'_{\langle 3\rangle}$ respectively, such that $t = \tau'_{\langle 1\rangle}[\tau_{\langle 1\rangle}] = \tau'_{\langle 3\rangle}[s^i_{\langle 3\rangle}]$ (via $pop\text{-}in_{\langle 2\rangle}\&copy\text{-}in_{\langle 1,3\rangle}$).

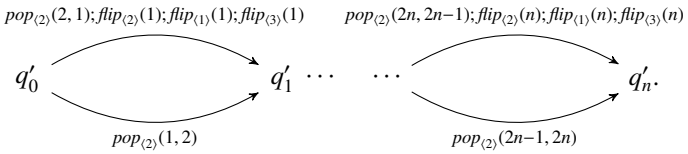
[7] From this point forward we generally refrain from giving explicit names to auxiliary states, for simplicity. Such names can be given, similarly as in the push-mode case, in order to show that Q' has size polynomial in the size of $\mathcal{M}$.

- Push $\tau_{I\langle 2\rangle}$ back on the stack (via $push_{\langle 2\rangle}$).
- Then make a guess $\tau''_{\langle 1\rangle}$ of $\tau_{\langle 1\rangle}$ and push it on the stack. While doing so, update $\tau'_{\langle 3\rangle}$ to the mask $s^{i'}_{\langle 3\rangle}$ satisfying $\tau'_{\langle 1\rangle}[\tau''_{\langle 1\rangle}] = \tau'_{\langle 3\rangle}[s^{i'}_{\langle 3\rangle}]$ (via $guess_{\langle 1\rangle}\&copy\text{-}in_{\langle 3\rangle}$).
- Finally, push on the stack $s^{i'}_{\langle 3\rangle}\,\tau_{I\langle 2\rangle}$ ($push_{\langle 3,2\rangle}$).

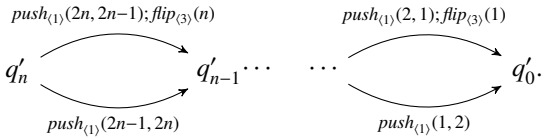
The $pop\text{-}in_{\langle 3\rangle}$ gadget simply performs a pop-into place for component $\langle 3\rangle$:



for designated $q'_0, \cdots, q'_n \in Q'$. Next, $pop\text{-}in_{\langle 2\rangle}\&copy\text{-}in_{\langle 1,3\rangle}$ is the following gadget, resembling the previously described $switch(\ldots, \uparrow)$.



Lastly, the $guess_{\langle 1\rangle}\&copy\text{-}in_{\langle 3\rangle}$ gadget is similar to $pop\text{-}in_{\langle 2\rangle}\&copy\text{-}in_{\langle 1,3\rangle}$, only that instead of reading (i.e. popping) a mask from the stack, it guesses and pushes it:



That is, it makes a guess for the mask at component $\langle 1\rangle$ and pushes it on the stack while updating component $\langle 3\rangle$.

In order to conclude the definition of δ', we need to add pop-transitions for consuming rows starting with the "exceptional" symbol $k+1$. As explained in the main text, the purpose of these transitions is simply to verify the continuity of the masks used in the $\langle 1\rangle$-component of $\mathcal{S}$, using also information stored in the two $\langle 3\rangle$-components of the given stack row. In particular, $\mathcal{S}$ must verify that those two $\langle 3\rangle$-components are equal. We therefore add the following transition sequences in $\mathcal{S}$,

$$(q, j, \uparrow) \xrightarrow{pop_{\langle 0\rangle}(k+1)} \xrightarrow{pop\text{-}in_{\langle 3\rangle}} \xrightarrow{clear_{\langle 2\rangle}} \xrightarrow{pop_{\langle 1\rangle}} \xrightarrow{clear_{\langle 2\rangle}} \xrightarrow{pop_{\langle 3\rangle}} (q, j, \uparrow)$$

for every $q \in Q$ and $j \in \{1, \cdots, n\}$.

We claim that $\mathcal{S}$ reaches q'_F iff $\mathcal{M}$ pops \$ from its stack. Note first that, if $\mathcal{M}$ has a computation leading to a pop-\$, then $\mathcal{S}$ can simulate it by making sure it makes all guesses right, and thus reach q'_F. Conversely, as argued above, $\mathcal{S}$

may only reach its final state if all its guesses on used masks are correct. But any such run would faithfully simulate a run of $\mathcal{M}$ leading to pop-\$, by construction. Finally, observe that the $\mathcal{M} \to \mathcal{S}$ reduction is poly-time. In particular, $\mathcal{S}$ has $6n + k + 1$ registers and a state space of size polynomial in the number of transitions in $\mathcal{M}$. □

The EXPTIME-hardness carries over to the language emptiness problem associated with infinite-alphabet pushdown automata [8] and register pushdown automata [22]. Since the latter allows for storage of identical values in different registers, their hardness can also be established more directly by encoding relative to two fixed data values for 0 and 1. These different policies for register management are known to lead to different complexity bounds for emptiness testing in the absence of pushdown store: NP-completeness [21][8] (injective assignment) vs PSPACE-completeness (non-injective assignment) [10]. Perhaps surprisingly, we have shown the presence of pushdown store cushions these differences and there is no analogous complexity gap.

5. Global Reachability

We now move on to investigate global reachability for r-PDRS. We show that, given an r-PDRS $\mathcal{S}$ and a representation C of a set of configurations of $\mathcal{S}$, one can construct, in exponential time, a representation of the set of configurations $\mathsf{Pre}^*_{\mathcal{S}}(C)$ from which $\mathcal{S}$ can reach a configuration in C. To that end we extend the methodology of Bouajjani, Esparza and Maler [5] to the infinite alphabet setting.

The developments in this section rely on an auxiliary variant of (stack-free) register automata which feature symbolic transitions representing multiple rearrangements of registers. In order to describe them, let us introduce ***r-register manipulations***, which are partial functions $R \in [r] \times [r] \hookrightarrow \{0, 1\}$ such that $R^{-1}\{1\}$ is a partial injection. We denote the set of all such partial functions by RegMan_r and use R^b to refer to $R^{-1}\{b\}$, for $b \in \{0, 1\}$. Given $R, S \in \mathsf{RegMan}_r$, we define $R \,;\, S$ as follows.

$$(R \,;\, S)(i,\ j) = \begin{cases} 1 & (S^1 \circ R^1)(i) = j \\ 0 & \exists k \in [r].\, (R^1(i) = k \wedge S^0(k) = j) \vee (R^0(i) = k \wedge S^1(k) = j) \end{cases}.$$

Moreover, given $i \in [r]$, we shall write $R_{i\bullet}$ for the partial function defined by:

$$R_{i\bullet}(j, i) = 0 \quad \text{for all } j \in [r], \qquad\qquad R_{i\bullet}(j, j) = 1 \quad \text{for all } j \neq i.$$

Register manipulations can be seen as abstract predicates on register assignments. In particular, given two register assignments τ, τ', we write $\tau\ R\ \tau'$ just if, for all $(i,\ j) \in \mathbf{dom}\, R$:

$$R(i,\ j) = 0 \text{ implies } \tau(i) \neq \tau'(j), \text{ and } R(i,\ j) = 1 \text{ implies } \tau(i) = \tau'(j).$$

[8] This result is affected by registers initially containing a special *undefined* value, without which the emptiness problem is reducible to that for finite automata and, consequently, NL-complete.

Definition 12. A ***register-manipulating r-register automaton*** (r-RMRA) is a tuple $\langle Q, F, \Delta\rangle$ with Q a finite set of states, $F \subseteq Q$ a subset of final states and $\Delta \subseteq Q \times \mathsf{OP}_r \times Q$ the transition relation, with $\mathsf{OP}_r = [r] \cup \{\bullet\} \cup \mathsf{RegMan}_r$. A ***configuration*** is a pair (q, τ) consisting of a state q and an r-register assignment τ. We say that a configuration (q_2, τ_2) is an *x-accepting successor* of a configuration (q_1, τ_1) and write $(q_1, \tau_1) \vdash^x (q_2, \tau_2)$ just if there is some transition $(q_1, o, q_2) \in \Delta$, $x \in \mathcal{D} \cup \{\varepsilon\}$ and either:

- $o = i$ for some $i \in [r]$, $\tau_1 = \tau_2$ and $\tau_1(i) = x$
- $o = \bullet$, $\tau_1 = \tau_2$ and $x \notin \nu(\tau_1)$.
- $o = R$ for some $R \in \mathsf{RegMan}_r$, $\tau_1\ R\ \tau_2$ and $x = \varepsilon$.

By some abuse of notation, we will write $(q, \tau) \vdash^w (q', \tau')$ to denote the existence of a (possibly empty) sequence of configurations $(q_0, \tau_0) \vdash^{x_1} (q_1, \tau_1) \vdash^{x_2} \cdots \vdash^{x_n} (q_n, \tau_n)$ for which $(q_0, \tau_0) = (q, \tau)$, $(q_n, \tau_n) = (q', \tau')$ and $w = x_1 \cdots x_n$. If $(q, \tau) \vdash^w (q', \tau')$ for some τ' and $q' \in F$ then we say that w is ***accepted*** from (q, τ).

The operations of RMRAs generalise the stack-free operations of PDRSs: $i \in [r]$ specifies reading a name already present in the i-th register, $\bullet$ reads a locally fresh name and $R \in \mathsf{RegMan}_r$ is an internal action such that if $q \xrightarrow{R} q'$ then any configuration (q, τ) may transition to any configuration (q, τ') satisfying $\tau\ R\ \tau'$. In what follows, we will start RMRAs from various initial configurations, so we do not include an initial state or register assignment in their specifications.

Definition 13. Given an r-RMRA $\mathcal{A} = \langle Q, F, \Delta\rangle$, a state $q \in Q$ and an r-register assignment τ, we set:

$$\mathcal{L}(\mathcal{A})(q, \tau) = \{w \in \mathcal{D}^* \mid w \text{ is accepted by } \mathcal{A} \text{ from } (q, \tau)\}.$$

Moreover, given an r-PDRS $\mathcal{S} = \langle P, q_I, \tau_I, \delta\rangle$ such that $P \subseteq Q$, we say that $\mathcal{A}$ ***represents*** the $\mathcal{S}$-configuration (p, τ, s) whenever $s \in \mathcal{L}(\mathcal{A})(p, \tau)$. We write $C(\mathcal{A})$ for the set of $\mathcal{S}$-configurations represented by $\mathcal{A}$.

Given an r-RMRA characterising a set of configurations of an r-PDRS $\mathcal{S}$, our aim is to construct another RMRA that represents exactly those configurations of $\mathcal{S}$ that can reach configurations in $C(\mathcal{A})$, i.e. we aim to construct a representation of $\mathsf{Pre}^*_{\mathcal{S}}(C(\mathcal{A}))$.

We shall do this in the "saturation" style of the classical construction of [5] but we need more notation in order to deal with the infinite alphabet. Given $R \in \mathsf{RegMan}_r$, we say that:

- R is *consistent* with the statement $i = j$ just if $R(i, j) \neq 0$ and $[i \in \mathbf{dom}\, R^1 \vee j \in \mathbf{ran}\, R^1]$ implies $R^1(i) = j$, and in that case we write $R \parallel i{=}j$. (resp. $R \parallel i^\bullet$).
- R is consistent with $i^\bullet$ just if $i \notin \mathbf{dom}\, R^1$, and in that case we write $R \parallel i^\bullet$.

So, the meaning of $R \parallel i^\bullet$, is that i in the situation before R may be locally fresh with respect to the situation after R. If $R \parallel i{=}j$ (resp. $R \parallel i^\bullet$) then we write $R[i = j]$ (resp. $R[i^\bullet]$) for $R \cup \{(i, j) \mapsto 1\}$ (resp. $R \cup \{(i, j) \mapsto 0 \mid j \in [r]\}$). Note the difference between $R_{i^\bullet}$ and $R[i^\bullet]$.

We write $q \xrightarrow{R}{}^{*} q'$ just if there is some finite, possibly empty, sequence $\langle q_i \rangle_{i \in [n]}$ such that $q_1 = q$ and $q_n = q'$ and, for all $i \in [n-1]$, $q_i \xrightarrow{R_i} q_{i+1}$ and $R_1 \,;\, \cdots \,;\, R_{n-1} = R$.

Definition 14. Given an r-PDRS $\mathcal{S}$ over states P, and given an r-RMRA $\mathcal{A}$ over states Q and transitions Δ, such that $P \subseteq Q$ and Δ contains no transitions to states in P, we construct another r-RMRA $\mathsf{SAT}(\mathcal{A})$ by induction (note that op ranges over OP_r):

$$\frac{p \xrightarrow[\mathcal{A}]{op} p'}{p \xrightarrow[\mathsf{SAT}(\mathcal{A})]{op} p'}\,(N) \qquad \frac{p \xrightarrow[\mathcal{S}]{i^\bullet} p'}{p \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R_i \bullet} p'}\,(i) \qquad \frac{p \xrightarrow[\mathcal{S}]{push(i)} p' \;\; p' \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R}{}^{*} q \;\; q \xrightarrow[\mathsf{SAT}(\mathcal{A})]{j} q'}{p \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R[i=j]} q'}\,(ii)$$

$$\frac{p \xrightarrow[\mathcal{S}]{push(i)} p' \;\; p' \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R}{}^{*} q \;\; q \xrightarrow[\mathsf{SAT}(\mathcal{A})]{\bullet} q'}{p \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R[i^\bullet]} q'}\,(iii) \qquad \frac{p \xrightarrow[\mathcal{S}]{pop(i)} p'}{p \xrightarrow[\mathsf{SAT}(\mathcal{A})]{i} p'}\,(iv) \qquad \frac{p \xrightarrow[\mathcal{S}]{pop^\bullet} p'}{p \xrightarrow[\mathsf{SAT}(\mathcal{A})]{\bullet} p'}\,(v)$$

where we additionally require $R \parallel i{=}j$ in rule (ii), and $R \parallel i^\bullet$ in rule (iii).

The above construction can be carried out in exponential time: consider that there are at most $|Q \times \mathsf{OP}_r \times Q|$ many transitions added, which is at most exponential in the size of the input. For each transition, computation is either trivial or, in (*ii*) and (*iii*), involves computing exponentially many graph reachability queries.

Theorem 15. *Given r-PDRS $\mathcal{S}$ and r-RMRA $\mathcal{A}$ as above, $C(\mathsf{SAT}(\mathcal{A})) = \mathsf{Pre}^*_{\mathcal{S}}(C(\mathcal{A}))$.*

We show the theorem in two parts. First we show soundness:

Lemma 16. *If $(p, \tau, w) \vdash^* (p', \tau', w')$ and $w' \in \mathcal{L}(\mathcal{A})(p', \tau')$ then $w \in \mathcal{L}(\mathsf{SAT}(\mathcal{A}))(p', \tau')$.*

Proof. Let the witness to the first premise be k, the proof is by induction on k. When $k = 0$ the result follows from rule (N). When $k = n + 1$, the transition sequence has the form:

$$(p, \tau, w) \vdash (p'', \tau'', w'') \vdash^n (p', \tau', w').$$

It follows from the induction hypothesis that $w'' \in \mathcal{L}(\mathsf{SAT}(\mathcal{A}))(p'', \tau'')$ (*). We continue by analysing the initial transition.

- If the initial transition is by $p \xrightarrow{i^\bullet} p''$ then $\tau'' = \tau[i \mapsto a]$ and $w'' = w$ for some a fresh for τ. By part (i) of the construction also $(p, \tau) \vdash (p'', \tau'')$ and hence $w \in \mathcal{L}(\mathsf{SAT}(\mathcal{A}))(p, \tau)$.
- If the initial transition is by $p \xrightarrow{push(i)} p''$ then $\tau = \tau''$ and $w'' = \tau(i)w$. By (*) there must be some transition sequence $(p'', \tau) \vdash^* (q, \sigma) \vdash (q', \sigma)$ in $\mathsf{SAT}(\mathcal{A})$ with the final transition being the only transition to read an input: $\tau(i)$, and $w \in \mathcal{L}(\mathsf{SAT}(\mathcal{A}))(q', \sigma)$. This final transition must be justified either by (1) $q \xrightarrow{j} q'$ or (2) $q \xrightarrow{\bullet} q'$. In both cases, the initial part of the sequence must be justified by some register manipulations $p'' \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R}{}^{*} q$. We distinguish between the two cases:

(1) In the first case, necessarily $\tau(i) = \sigma(j)$ and, therefore, $R \parallel i = j$. It follows from part (ii) of the construction that $(p, \tau) \vdash (q', \sigma)$ and hence $w \in \mathcal{L}(\mathsf{SAT}(\mathcal{A}))(p, \tau)$.

(2) In the second case, necessarily $\tau(i) \notin \nu(\sigma)$ and, therefore, $R \parallel i^\bullet$. It follows from part (iii) of the construction that $(p, \tau) \vdash (q', \sigma)$ and hence $w \in \mathcal{L}(\mathsf{SAT}(\mathcal{A}))(p, \tau)$.

- If the initial transition is by $p \xrightarrow{pop(i)} p''$ then $\tau = \tau''$ and $w = \tau(i)w''$ and it follows from part (iv) of the construction that $w \in \mathcal{L}(\mathsf{SAT}(\mathcal{A}))(p, \tau)$.
- If the initial transition is by $p \xrightarrow{pop^\bullet} p''$ then $\tau = \tau''$ and $w = aw''$ for some a fresh for τ. It follows from part (v) of the construction that $w \in \mathcal{L}(\mathsf{SAT}(\mathcal{A}))(p, \tau)$.

□

Corollary 17. $\mathsf{Pre}^*_{\mathcal{S}}(C(\mathcal{A})) \subseteq C(\mathsf{SAT}(\mathcal{A}))$.

For completeness, i.e. to see that $\mathsf{SAT}(\mathcal{A})$ does not accept any word which is not the stack component of some configuration reaching the target set, we first generalise the quantity $\mathsf{Pre}^*_{\mathcal{S}}(C(\mathcal{A}))$. Define $\mathsf{P}^*_{\mathcal{S}}(\mathcal{A})$ as follows:

$$\mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q, \tau) = \begin{cases} \mathcal{L}(\mathcal{A})(q, \tau) & q \notin P \\ \{s \mid (q, \tau, s) \in \mathsf{Pre}^*_{\mathcal{S}}(C(\mathcal{A}))\} & q \in P \end{cases}$$

(recall that P is the set of states of the PDRS $\mathcal{S}$). Observe that, if $q \in P$ and $w \in \mathcal{L}(\mathcal{A})(q, \tau)$ then $(q, \tau, w) \in C(\mathcal{A})$, which implies $w \in \{s \mid (q, \tau, s) \in \mathsf{Pre}^*_{\mathcal{S}}(C(\mathcal{A}))\}$.

Lemma 18. *Transitions in* $\mathsf{SAT}(\mathcal{A})$ *preserve* $\mathsf{P}^*_{\mathcal{S}}(\mathcal{A})$*:*

- *If* $q \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R} q'$, $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q', \tau')$ *and* $\tau \; R \; \tau'$ *then* $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q, \tau)$.
- *If* $q \xrightarrow[\mathsf{SAT}(\mathcal{A})]{i} q'$ *and* $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q', \tau)$ *then* $\tau(i)w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q, \tau)$.
- *If* $q \xrightarrow[\mathsf{SAT}(\mathcal{A})]{\bullet} q'$, $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q', \tau)$ *and* $a \notin \nu(\tau)$ *then* $aw \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q, \tau)$.

Proof. By induction on the construction of $\mathsf{SAT}(\mathcal{A})$ and case analysis on the last rule used in it:

(N) Note that, by our initial assumption, $\mathcal{A}$ contains no incoming transitions to states in P. Thus, in every case, $q' \notin P$. Suppose now $q \xrightarrow[\mathcal{A}]{R} q'$, $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q', \tau')$ and $\tau \; R \; \tau'$. Then, $w \in \mathcal{L}(A)(q', \tau')$ and therefore, since $(q, \tau) \xrightarrow{\epsilon} (q', \tau')$ in $\mathcal{A}$, $w \in \mathcal{L}(A)(q, \tau)$. Thus, using also our previous observation, $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q, \tau)$. The other cases are shown in a similar manner.

(i) If the construction is concluded by (i) then assume $p \xrightarrow[\mathcal{S}]{i^\bullet} p'$ and let $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p', \tau')$ and $\tau \; R_{i^\bullet} \; \tau'$, with $p = q$. It follows from the assumption that $(p, \tau, w) \vdash (p', \tau', w)$ whenever $\tau' = \tau[i \mapsto a]$ for some fresh a. Since this is true of any τ such that $\tau \; R_{i^\bullet} \; \tau'$ the result follows.

(ii) Assume $p \xrightarrow[\mathcal{S}]{push(i)} p'$, also $p' \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R}{}^{*} q''$, $R \parallel i = j$ and $q'' \xrightarrow[\mathsf{SAT}(\mathcal{A})]{j} q'$, with $p = q$. Let $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q', \tau')$ and let $\tau\ R[i = j]\ \tau'$. It follows from the induction hypothesis that $\tau'(j)w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q'', \tau')$ and, since $R[i = j]$ is stronger than R, that $\tau'(j)w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p', \tau)$. Since $\tau(i) = \tau'(j)$, it follows from the first assumption that $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p, \tau)$.

(iii) Assume $p \xrightarrow[\mathcal{S}]{push(i)}$, $p' \xrightarrow[\mathsf{SAT}(\mathcal{A})]{R}{}^{*} q''$, $R \parallel i^\bullet$ and $q'' \xrightarrow[\mathsf{SAT}(\mathcal{A})]{\bullet} q'$. Let $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q', \tau')$ and $\tau\ R[i^\bullet]\ \tau'$. Since $\tau(i)$ is fresh for τ', it follows from the induction hypothesis that $\tau(i)w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(q'', \tau')$ and, since $R[i^\bullet]$ is stronger than R, that $\tau(i)w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p', \tau)$. Finally, it follows from the first assumption that $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p, \tau)$.

(iv) Assume $p \xrightarrow[\mathcal{S}]{pop(i)} p'$. Let $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p', \tau)$. Then by definition $\tau(i)w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p, \tau)$.

(v) Assume $p \xrightarrow[\mathcal{S}]{pop^\bullet} p'$. Let $w \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p', \tau)$ and let a be fresh for τ. Then by definition $aw \in \mathsf{P}^*_{\mathcal{S}}(\mathcal{A})(p, \tau)$.

□

Corollary 19. $C(\mathsf{SAT}(\mathcal{A})) \subseteq \mathsf{Pre}^*_{\mathcal{S}}(C(\mathcal{A}))$.

Proof. Let $(p, \tau) \vdash \cdots \vdash (q, \tau')$ with $p \in P$ be a transition sequence of $\mathsf{SAT}(\mathcal{A})$ accepting some word w. Since q is necessarily a final state, $\epsilon \in \mathcal{L}(\mathcal{A})(q, \tau')$ and then it follows from Lemma 18 that $w \in \mathsf{Pre}^*_{\mathcal{S}}(C(\mathcal{A}))$. □

We can thus verify whether one can reach a configuration represented by $\mathcal{A}$ from a given configuration: construct the corresponding $\mathsf{SAT}(\mathcal{A})$ and check membership. To implement the latter in nondeterministic space, given a source configuration (q, τ, w), we need $O(\log |Q_{\mathsf{SAT}(\mathcal{A})}| + p(r) + \log |w|)$ bits to track the state, register assignment and position in w respectively. This is polynomial space in $\mathcal{S}, \mathcal{A}, w$ which, along with the construction of $\mathsf{SAT}(\mathcal{A})$, yields an exponential-time reachability testing routine.

Finally, let us observe that RMRAs are no more expressive than register automata with nondeterministic reassignment [14]. An r-RMRA $\mathcal{A} = \langle Q, F, \Delta \rangle$ can be seen as an *r-register automaton with nondeterministic reassignment* (r-RA$_{nr}$) if $\Delta \subseteq Q \times \mathsf{OP}^-_r \times Q$, with $\mathsf{OP}^-_r = [r] + \{ R_{i^\bullet} \mid i \in [r] \}$. The proof of this result is contained in Appendix A.

Lemma 20. *For any r-RMRA $\mathcal{A}$, one can construct a* $(2r{+}1)$*-RA$_{nr}$ $\hat{\mathcal{A}}$ such that, for each $\mathcal{A}$-configuration κ there exists a $\hat{\mathcal{A}}$-configuration $\hat{\kappa}$ satisfying $\mathcal{L}(\mathcal{A})(\kappa) = \mathcal{L}(\hat{\mathcal{A}})(\hat{\kappa})$.*

6. Higher-Order Pushdown Systems

We now consider reachability at higher orders, defining pushdown register automata as a register-equipped analogue of the classical definition of [15]. We show that the reachability problem is undecidable.

A 1-*stack* is just a finite sequence of elements of $\mathcal{D}$. For $n > 1$, an n-*stack* is a finite sequence of $n{-}1$-stacks. We consider the following operations on 1-*stacks*:

- $push^a_1\langle a_l, \ldots, a_1 \rangle = \langle a, a_l \ldots, a_1 \rangle$ for any $a \in \mathcal{D}$,
- $pop_1\langle a_l, a_{l-1}, \ldots, a_1 \rangle = \langle a_{l-1}, \ldots, a_1 \rangle$,

- $top_1\langle a_l, a_{l-1}, \ldots, a_1\rangle = a_l$,

and, in connection with n-stacks for $n > 1$:

- $push_1^a\langle s_l, \ldots, s_1\rangle = \langle push_1^a\ s_l, s_{l-1}, \ldots, s_1\rangle$,
- $push_k\langle s_l, \ldots, s_1\rangle = \langle push_k\ s_l, s_{l-1}, \ldots, s_1\rangle$ if $2 \leq k < n$,
- $push_k\langle s_l, \ldots, s_1\rangle = \langle s_l, s_l, \ldots, s_1\rangle$ if $k = n$,
- $pop_k\langle s_l, \ldots, s_1\rangle = \langle pop_k\ s_l, s_{l-1}, \ldots, s_1\rangle$ if $1 \leq k < n$,
- $pop_k\langle s_l, \ldots, s_1\rangle = \langle s_{l-1}, \ldots, s_1\rangle$ if $k = n$,
- $top_1\langle s_l, \ldots, s_1\rangle = top_1\ s_l$,

noting that every operation except $push_1^a$ is undefined when applied to an empty stack. Finally, we write $\langle\,\rangle_k$ for the k-stack defined as ϵ when $k = 1$ and $\langle\langle\,\rangle_{k-1}\rangle$ otherwise.

Definition 21. An ***order-n pushdown r-register system*** (r-nPDRS) is an r-PDRS with the vocabulary of operations Op_r extended in the following way:

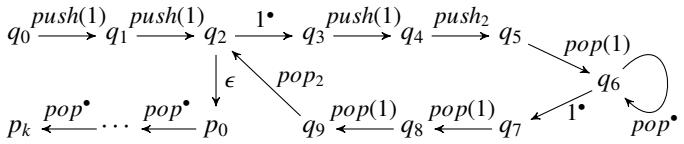
$$Op_r^n = Op_r \cup \{\, push_k,\ pop_k \mid 2 \leq k \leq n \,\}.$$

A configuration of an r-nPDRS is a triple $(q,\ \tau,\ s)$ with q and τ as before and s now an n-stack. The initial configuration is $(q_I,\ \tau_I,\ \langle\,\rangle_n)$. A configuration $(q_2,\ \tau_2,\ s_2)$ is said to be a successor of a configuration $(q_1,\ \tau_1,\ s_1)$ just if there is some $op \in Op_r^n$ such that $(q_1,\ op,\ q_2) \in \delta$ and one of the following is true:

- $op = i^\bullet$, $\forall j.\, \tau_2(i) \neq \tau_1(j)$, $\forall j \neq i.\, \tau_2(j) = \tau_1(j)$ and $s_1 = s_2$,
- $op = push(i)$, $\tau_2 = \tau_1$ and $s_2 = push_1^{\tau_1(i)}\ s_1$,
- $op = pop(i)$, $\tau_2 = \tau_1$, $top_1\ s_1 = \tau_1(i)$ and $s_2 = pop_1\ s_1$,
- $op = pop^\bullet$, $\tau_2 = \tau_1$, $\forall j.\, \tau_1(j) \neq top_1\ s_1$ and $s_2 = pop_1\ s_1$,
- $op = push_k$, $k > 1$, $\tau_2 = \tau_1$ and $s_2 = push_k\ s_1$,
- $op = pop_k$, $k > 1$, $\tau_2 = \tau_1$ and $s_2 = pop_k\ s_1$.

We show that, for all r and $n > 1$, r-nPDRS have undecidable reachability problems by showing undecidability for $r = 1$ and $n = 2$. For 1-2PDRS, we will write a configuration $(q,\ \{1 \mapsto a\},\ s)$ generally as $(q,\ a,\ s)$. The following example shows how data held on a 1-stack of a 1-2PDRS can be copied and interrogated.

Example 22. We demonstrate the lack of a uniform bound on the number of distinct data values needed to reach a designated state (for r-PDRS that bound is $3r$). For every $k \in \mathbb{N}$, there is an 1-2PDRS needing more than k names in order to reach state p_k.



The idea is as follows, let the initial register assignment be the single element #. Whenever the machine is in state q_2, its 2-stack is of the form $\langle\langle a_m, \ldots, a_1, \#, \#\rangle\rangle$, for $m \geq 0$, with $a_i \neq a_j \neq \#$ for all $i \neq j$. The use of ## serves to mark out the bottom of the stack. On each iteration of the cycle starting in q_2, an additional data value is pushed onto the singleton 1-stack (upon leaving state q_3) which is then verified to be different from all the others. This verification is implemented by first taking a copy of the 1-stack using $push_2$, then checking that the data value in the register is different from all other values on the stack using $pop^\bullet$. Now, the top copy of the 1-stack will be exhausted and the machine simply discards it with pop_2, restoring the invariant and returning to state q_2. Finally, note that the automaton can transition from q_2 to p_k only if it has gathered at least k non-# values in its stack.

To show the undecidability of the reachability problem for higher-order PDRS, we reduce from the emptiness problem for weak pebble automata, which is known to be undecidable [20, 23]. We find it convenient to use pebbles, because the push and pop instructions have a direct analogue in placing and lifting a pebble.

Theorem 23. *The reachability problem for r-nPDRS is undecidable for any $n > 0$.*

Proof. For the full definition of pebble automata we direct the reader to [20]; we here recall only an outline. A *weak k-pebble automaton* (k-PA) is a tuple (Q, q_0, F, T) where Q is a finite set of states of which q_0 is initial and $F \subseteq Q$ are final, and T is a finite set of transitions. Each transition is of the form $\alpha \to \beta$. In general α has shape (i, d, P, V, q) or (i, P, V, q) specifying: the index of the head pebble; (possibly) the data value under the head; the set of pebbles whose position coincides with the head; the set of pebbles on the same data as the head; and the state respectively. The shape of β is (q, A) with q the state to be moved to and $A \in \{\text{stay, right, place, lift}\}$ the pebbling action to perform. Given a word w, a configuration of such a machine on w is a tuple (i, q, θ) where $\theta : \{1, \ldots, i\} \to \{1, \cdots, |w|\}$ specifies the locations of the currently placed pebbles. WLOG we assume that $|w| > 0$. The initial configuration is $(1, q_0, \theta_0)$ with $\theta_0(1) = 1$.[9]

We now consider a particular kind of data structure which will be very useful for the undecidability argument.

Indexed stacks. We say that a 1-stack is in *m-padded index* form just if it has the following shape:

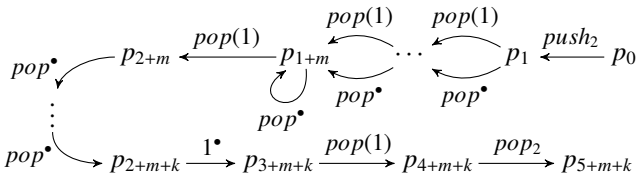
$$c_m \cdots c_1 \; a_n \, b_n \cdots a_1 \, b_1 \, \#\#$$

in which $\#, a_j, b_j, c_j \in \mathcal{D}$ for all j. Furthermore, for all j and k: $a_j \neq \#$, $a_j \neq b_k$, and $a_j = a_k \implies j = k$. Also, for all j there is j' such that $c_j = a_{j'}$. Such 1-stacks are composed of three segments. The first segment ## is a 'bottom of stack'

[9] Our definition differs slightly from [20] in that the latter uses θ with range $\{0, \cdots, |w| + 1\}$, with positions 0 and $|w| + 1$ corresponding to end-markers. Here we do not treat end-markers, hence the different range for θ. End-markers can be treated in the same way as constants (cf. the following section).

marker (such as in Example 22). The second segment $a_n b_n \cdots a_1 b_1$ represents the indexed word $(a_1, b_1) \cdots (a_n, b_n)$, where the names $a_1, \cdots, a_n$ are called the *indices* and $b_1, \cdots, b_n$ are the *values* of the word. Finally, the third segment is a sequence of m indices.

Stacks in padded index form support a kind of dereferencing operation which allows a 1-2PDRS to retrieve the data value stored at some constant offset k from one of the indices. For m-padded index stacks the operation is implemented by the following gadget, which we call $deref(m, k)$.



Starting from state p_0 with a_j in its register, the machine first saves a copy of the working 1-stack. In the current copy, it discards the top segment and then the segment $a_n b_n \cdots a_{j+1} b_{j+1}$. It then transitions to state p_{2+m} (consuming a_j) and then discards the next k-elements of the stack. Having traversed the constant offset it then refreshes the contents of its register and verifies that the new assignment is exactly the element on top of its 1-stack. Finally, it restores the original 1-stack.

The following result is easily verified by inspecting the construction.

Lemma 24. *Let $(p_0, a, \langle s \rangle)$ be a configuration of an 1-2PDRS containing $deref(m, k)$ and in which s is an m-padded index with $s(i) = a$ an index, for some i.*[10] *Then $(p_0, a, \langle s \rangle) \vdash^* (p_{5+m+k}, b, s')$ iff $b = s(i + k + 1)$ and $s' = \langle s \rangle$.*

Using *deref* it is possible to implement useful operations associated with the index structure. In particular, with index a assigned to the register, $lookup(m) = deref(m, 0)$ looks up the value associated with index a and $pred(m) = deref(m, 1)$ finds the preceding index of a.

6.1. Simulation of k-PA without constants

Given a k-PA $\mathcal{A} = (Q, q_0, F, T)$, we construct a 1-2PDRS $\mathcal{S} = \langle Q', q_I, \tau_I, \delta \rangle$ that first guesses a non-empty[11] word w and then checks that $w \in \mathcal{L}(\mathcal{A})$ by simulating an accepting run of $\mathcal{A}$ on w.

For clarity of exposition, we shall assume that $\mathcal{A}$ does not recognise constants: i.e. there are no transitions of the form $(i, d, P, V, q) \to \beta$. As we demonstrate in the following section, the proof can be extended to account for such transitions by modelling constants as sequences of identical data values of a fixed length.

The state space Q' of $\mathcal{S}$ consists of the set

$$\{(q, j) \mid q \in Q \text{ and } 1 \leq j \leq k\}$$

[10] Here we write $s(i)$ for the i-th element of the 1-stack s; $s(0) = top_1\, s$.

[11] Given PA $\mathcal{A}$, it is straightforward to construct PA $\mathcal{A}'$ such that $\mathcal{A}'$ accepts a word iff $\mathcal{A}'$ accepts a non-empty word and $\mathcal{L}(\mathcal{A}) = \emptyset$ iff $\mathcal{L}(\mathcal{A}') = \emptyset$.

of *primary states*, the set $\{q^t_{(P,V)} \mid t = (i, P, V, q) \to \beta \in T\}$ and the set $\{q^t_{(q',A)} \mid t = \alpha \to (q', A) \in T\}$; as well as a number of auxiliary states which will be specified implicitly in the description to follow.

We say that a configuration κ of $\mathcal{S}$ is *proper* just if it is of the form:

$$((q,\, m),\, d,\, \langle\langle c_m, \ldots, c_1, a_n, b_n, \ldots, a_1, b_1, \#, \#\rangle\rangle)$$

with $w = b_n \cdots b_1$. We map proper configurations κ of $\mathcal{S}$ to configurations of $\mathcal{A}$ on w by the following surjection:

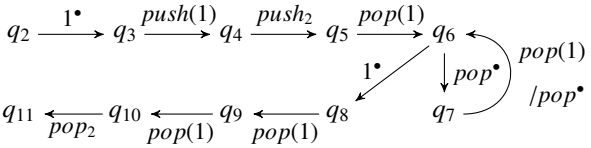
$$[\![\kappa]\!] = (m,\, q,\, \{j \mapsto n - i + 1 \mid 1 \leq j \leq m \wedge c_j = a_i\}).$$

In other words, proper configurations have m-padded indexed stacks where the index structure represents the input word, and the padding records the positions of the m (so far) placed pebbles.

The computation of $\mathcal{S}$ consists of two parts. First an initialisation phase computes the initial guess of the word w and puts $\mathcal{S}$ into a proper configuration κ with $[\![\kappa]\!] = (1,\, q_0,\, \theta_0)$ running on w. The second part is a loop in which the machine guesses an applicable $\mathcal{A}$-transition $\alpha \to \beta$ and simulates it.

Initialisation. Starting from its initial configuration, $\mathcal{S}$ begins in the same way as the system from Example 22 by pushing its initial register assignment twice to form a 'bottom of stack' marker. Then, starting from state q_2, it loops nondeterministically building an indexed representation of the word w on its top 1-stack. On each iteration, when the system is in state q_2, its 2-stack has shape $\langle\langle a_l, b_l, \cdots, a_1, b_1, \#, \#\rangle\rangle$ with singleton 1-stack in 0-padded index form.

From state q_2, $\mathcal{S}$ can either choose a new data value b_{l+1} followed by a new index a_{l+1} and return to state q_2, or it can transition to state $(q_0,\, 1)$ as described below. In the first case, the machine uses two gadgets to ensure that the choices respect the desired invariant. We describe the gadget that ensures that the choice of b_{l+1} is different from any index below; the gadget to ensure that a_{l+1} is different from all other stack elements is exactly as in Example 22.



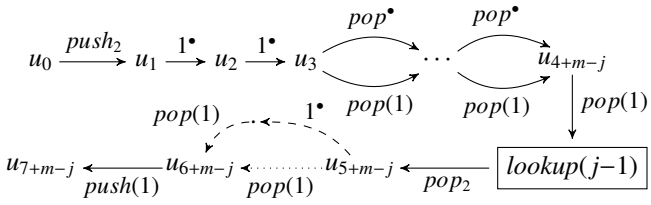
The path from q_2 to q_4 guesses the next data value x_{l+1} and places it at the top of the 1-stack. The remainder of the gadget is used to ensure that the guess does not coincide with one of the indices. The machine first takes a copy of the current 1-stack and then checks that every other data value on the stack is different from x_{l+1}. Finally, it restores the original 1-stack.

After iterating the loop on q_2 some number of times and thus choosing the word w, the machine then makes an extra copy of the top of the 1-stack and enters state $(q_0,\, 1)$. It follows from this construction that, for all d and s: $(q_I,\, \#,\, \langle\,\rangle_2) \vdash^* ((q_0,\, 1),\, d,\, s)$ iff there exists some word w such that $[\![((q_0,\, 1),\, d,\, s)]\!] = (1,\, q_0,\, \theta_0)$ running on w.

Simulation loop. At the start of each iteration of the simulation loop, $\mathcal{S}$ is in some proper configuration $\kappa = ((q,\, m),\, \langle s\rangle)$ with $s = \langle c_m, \ldots, c_1, a_n, b_n, \ldots, a_1, b_1, \#, \#\rangle$. It guesses an applicable transition $t = (m,\, P,\, V,\, q) \to (q',\, A)$

and moves to state $q^t_{(P,V)}$. To verify the applicability of the transition $\mathcal{S}$ must check the applicability of the P and V components.

To check applicability with respect to the V-component, $\mathcal{S}$ needs to check that V contains precisely those pebbles in positions which contain the same data value as that of the mth pebble. Put otherwise, assuming $c_m = a_{m'}$, we must have $V = \{j \mid \forall j'.\, c_j = a_{j'} \implies b_{j'} = b_{m'}\}$. Thus, for each $1 \leq j < m$, $\mathcal{S}$ will find the j' such that $c_j = a_{j'}$, and then check the condition: $b_{j'} = b_{m'} \iff j \in V$. Thus, $\mathcal{S}$ first copies the head position (currently at the top of the stack) into its register by executing two $1^\bullet$ operations followed by $pop(1)$ and $push(1)$. Note that, because $1^\bullet$ will always choose a data value that is fresh with respect to the current register contents, two occurrences of this operation are used sequentially to ensure that the correct data value can be guessed, even if it is already stored in the register. It then uses a copy of the $lookup(j)$ gadget to retrieve the data value indexed by the head pebble and performs $push(1)$ to put the data at the top of the stack. Then, for each $1 \leq j < m$ in sequence, $\mathcal{S}$ uses a copy of the following gadget to check that the value indexed by c_j (the data value under pebble j) is the same as that at the top of the stack just if $j \in V$.



This gadget copies the current 1-stack and then guesses and checks the position c_j of pebble j. It then uses a copy of the $lookup(j-1)$ gadget to store the data value indexed by pebble j in its register. Next it restores the old 1-stack (which has the data value under the head pebble at the top of the stack) and: if $j \in V$ then it verifies that the data value at the top and the data value in its register are the same (dotted arrow); if $j \notin V$ then it verifies that these data values are different (dashed arrows). At the end of the sequence of m copies of the gadget, $\mathcal{S}$ discards the redundant data value from the top of the stack.

Checking applicability with respect to the P component is much simpler: $\mathcal{S}$ needs to check that, for each $1 \leq j < m$, $c_j = c_m \iff j \in P$. This is achieved by the gadget:

$$u_0 \xrightarrow{push_2} u_1 \xrightarrow{1^\bullet} u_2 \xrightarrow{pop(1)} u_3 \xrightarrow{t_{m-1}} \cdots \xrightarrow{t_1} u_{2+m} \xrightarrow{pop_2} u_{3+m}$$

where each t_j is $pop(1)$ if $j \in P$, and $pop^\bullet$ otherwise.

At the end of both sequences of gadgets, $\mathcal{S}$ transitions to state $q^t_{(q',A)}$. By construction, we have $\kappa \vdash^* (q^\tau_{(q',A)}, d, s')$ iff $d = c_m$, $s' = \langle s \rangle$ and (m, P, V, q) is applicable to $[\![\kappa]\!]$.

In state $q^t_{(q',A)}$, $\mathcal{S}$ simulates the action of $\mathcal{A}$ executing A and then moves to simulating state q'. If A is stay, then $\mathcal{S}$ simply transitions to state (q', m). If A is lift-pebble, then $\mathcal{S}$ discards the top of the stack and transitions to $(q', m-1)$. If A is place-pebble then $\mathcal{S}$ pushes an extra copy of the top of the current 1-stack on to the top of the current 1-stack and transitions to $(q', m+1)$. Finally, if A is move right, $\mathcal{S}$ pops the current pebble position (top of the stack) into its

register and then copies the current 1-stack using $push_2$. It then uses a copy of the $pred(m-1)$ gadget to reassign to its register the next index in the sequence and then restores the saved 1-stack, pushes the assignment and transitions to state (q', m). The simulation loop then begins again.

It follows from the definition of the simulation loop that, for each proper configuration κ, and each configuration κ' in a primary state, $\kappa \vdash^* \kappa'$ iff κ' is also proper and $[\![\kappa]\!] \vdash^* [\![\kappa']\!]$. Hence, $\mathcal{L}(\mathcal{A})$ is non-empty iff there is a final state $q \in F$ and a pebble index m such that $\mathcal{S}$ has a run ending in state (q, m).

6.2. Simulation of k-PA with constants

Every pebble automaton induces a finite set of constants $\{c_1, \ldots, c_l\} \subset \mathcal{D} \cup \{\triangleright, \triangleleft\}$ which are those elements of $\mathcal{D}$ (and the start and end of word markers, where applicable) which are mentioned (by the s component) of transitions in T. Furthermore, the words accepted by a pebble automaton are delimited by markers $\triangleleft$, $\triangleright$. Both of these aspects, which were omitted from the treatment in the main text, are addressed here through an encoding of constants. Since 1-2PDRS has only one register, it does not have a native ability to recognise all such constants and so this must be encoded. However, it is straightforward to do so using fixed length sequences of identical data values.

To extend the construction from Theorem 23, we simulate constants appearing in the guessed word w. To this end, we consider an extended definition of m-padded index form stacks which have the following shape:

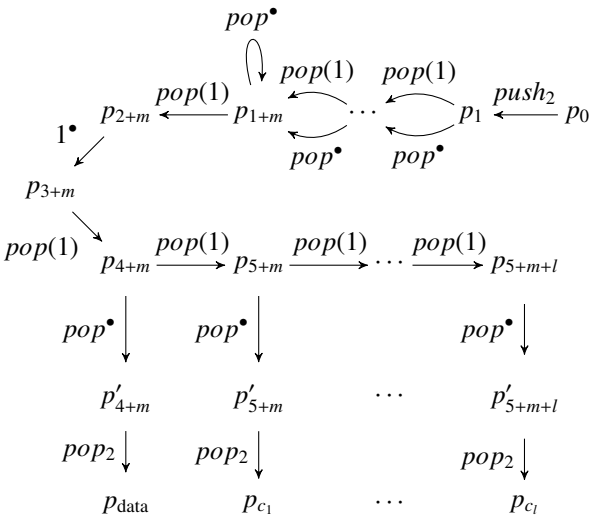
$$c_m \cdots c_1 a_n u_n \cdots a_1 u_1 \underbrace{\# \cdots \#}_{l+2 \text{ times}} .$$

Here, the individual data values b_j are replaced by words u_j of length $l + 1$. We require that each index a_j is distinct from every other as before and also that each a_j is distinct from any data value in any u_k. Each word u_j encodes either a particular constant or an anonymous data value. The encoding is as follows. We require each $u \in \{u_n, \ldots, u_1\}$ to be composed of two segments $u = b_i \cdots b_1 a_{l+1-i} \cdots a_1$ with $i > 0$, and $a_1 = \cdots = a_{l+1-i} \neq b_1 = \cdots = b_i$. The quantity i determines the value represented by the word:

$$[\![b_i \cdots b_1 a_{l+1-i} \cdots a_1]\!] = \begin{cases} b_i & \text{when } i = 1 \\ c_{i-1} & \text{otherwise} \end{cases} .$$

Also, the 'bottom of stack' marker has been lengthened to size $l + 2$ so that it is distinguished among the other segments. In this way it is effectively treated as the $l + 1$st constant.

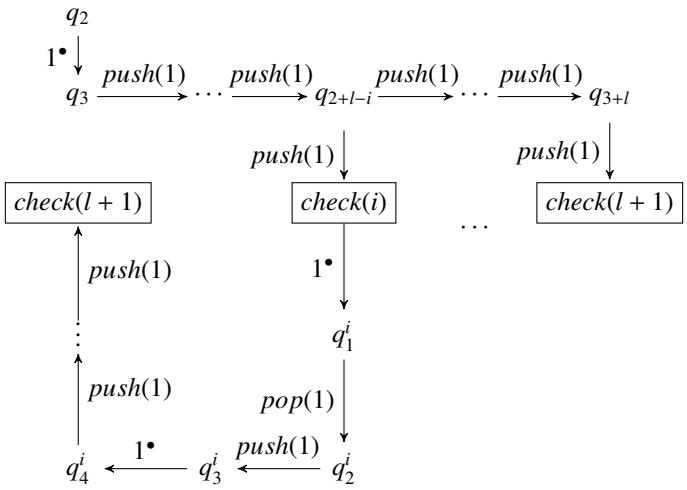
The operations over padded index stacks must be changed to reflect the more involved encoding. Both *lookup* and *pred* are much the same as before, but because *lookup* also needs to decode the value it finds at a given index before returning, the code for the two diverges. The extended $pred(m)$ gadget is simply $deref(m, l + 1)$, since the offset from one index to the next is now length $l + 1$. The new gadget for lookup is as follows.



This gadget starts in the same way as $deref(m, k)$ but, after popping off the desired index and reaching state p_{2+m}, it then proceeds to count the number of consecutive identical data values, thus establishing the number i. The result is recorded in its state space. Consumers of this operation, such as the simulation loop, can be straightforwardly (but tediously) modified to take into account whether or not a constant was discovered and do comparisons between values first according to the record in the state space and second (if both values are anonymous data) according to equality of data values.

When simulating transitions, $\mathcal{S}$ must only choose some applicable $(i, P, V, q) \to \beta$ if there are no other applicable transitions $(i', P', V', q') \to \beta'$. Observe that if configurations (i, P, V, q) and (i', d, P', V', q') are both applicable to the current configuration κ, then $i' = i$, $P' = P$, $V' = V$ and $q' = q$, since these are all determined by κ. So we modify $\mathcal{S}$ so that, when checking applicability of (i, P, V, q), it looks up and decodes (as above) the value (word of length $l + 1$) u under the head pebble and checks that it is not any constant d s.t. (i, d, P, V, q) is the antecedent of some other transition.

Finally, initialisation must be changed so that a word containing encoded constants can be guessed and stored as a padded index stack. We show the extended gadget for choosing a word u_{l+1} (having already chosen $u_1, \ldots, u_l$ for $l \geq 0$), the gadget for choosing a new index is similar to the one without constants.



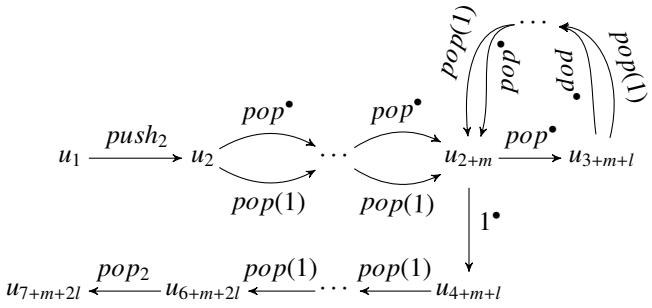
From state q_2 and with its 2-stack of shape $\langle\langle a_l, u_l, \ldots, a_1, u_1, \#, \ldots, \#\rangle\rangle$ $(0 \leq l)$, the top row of the gadget allows the machine to non-deterministically choose a new data value a, a number i and push i copies of a on to the top of the stack. It must then check that this data value a does not coincide with any of the existing indices $a_1, \ldots, a_l$ which is accomplished using a copy of the *check*(i) gadget, to be described below. At the end of the *check*(i) gadget, the stack of the machine is the same as at the start, i.e. it has the form:

$$\langle\langle \underbrace{a, \ldots, a}_{i \text{ times}}, a_l, u_l, \ldots, a_1, u_1, \underbrace{\#, \ldots, \#}_{l+2 \text{ times}}\rangle\rangle$$

and all that remains is to choose some b, push $l + 1 - i$ copies of b onto the stack and check that $a \neq b$ and that also b is different from any existing index. This is accomplished by the remaining part of the gadget.

In order to ensure that the word is properly delimited by the endmarkers, $\mathcal{S}$ is modified to always choose u_1 such that $[\![u_1]\!] = \triangleleft$ and to add an extra step of choosing a final encoded data value u_{n+1} and index a_{n+1} such that $[\![u_{n+1}]\!] = \triangleright$.

When started from a configuration whose top 1-stack is in extended m-padded index form, the *check*(m) gadget ensures that the data value in the machine's register is different from any index in the stack.



The gadget first preserves the working 1-stack and then discards the m-padding values. It then cycles, alternating between checking that an index is fresh for the value in its register and discarding a length $l + 1$ word u_j encoding a

data value or constant. At some point it guesses that it has met the bottom of the stack and verifies the guess. Finally it restores the preserved stack. □

7. Conclusion

We have studied reachability problems for pushdown register automata, adding new results to the literature on automata over infinite alphabets [22]. Although the automata are equipped with unbounded stack storage, we have shown that their discriminating power with respect to the infinite alphabet is quite restricted and only $3r$ elements suffice to answer reachability questions. Unlike in the pushdown-free case, variations in register disciplines no longer affect the complexity of reachability, which is EXPTIME-complete in every case. We have also shown that global reachability analysis can be carried out in exponential time. Finally, in contrast to the finite-alphabet case, the extension of pushdown storage to higher-order pushdown storage leads to undecidability.

Appendix A. From register-manipulating RAs to RAs with non-deterministic reassignment

We show the equivalence by a series of reductions, starting from RMRAs and reducing to more restricted machines until we reach RA_{nr}s.

Definition 25. An r-RA *of type X*, for $X \in \{\mathrm{I}, \mathrm{II}, \mathrm{III}\}$, is a triple $\mathcal{A} = \langle Q,\ F,\ \Delta\rangle$, $\Delta \subseteq Q \times \mathsf{OP}_r^X \times Q$, with:

(I) $\mathsf{OP}_r^{\mathrm{I}} = [r] \cup \{\bullet\} \cup \{\, R_{\alpha\bullet} \mid \alpha \subseteq [r]\,\}$, where $R_{\alpha\bullet}$ is the partial function given by, for all $i \in [r]$ and $j \in \alpha$, $R_{\alpha\bullet}(i, j) = 0$, and $R_{\alpha\bullet}(i, i) = 1$ for all $i \notin \alpha$.

(II) $\mathsf{OP}_r^{\mathrm{II}} = [r] \cup \{\bullet, \varepsilon\} \cup \{\, R_{i\bullet} \mid i \in [r]\,\}$.

(III) $\mathsf{OP}_r^{\mathrm{III}} = [r] \cup \{\varepsilon\} \cup \{\, R_{i\bullet}^{+} \mid i \in [r]\,\} \cup \{\, R_{i\bullet} \mid i \in [r]\,\}$.

RAs of types I and II are special cases of RMRAs. Type-III RAs are variants in which the semantics of transitions $q \xrightarrow{R_{i\bullet}^{+}} q'$ is identical to that of sequences of RMRA-transitions: $q \xrightarrow{R_{i\bullet}} \cdot \xrightarrow{i} q'$.

Let $\mathcal{A} = \langle Q,\ F,\ \Delta\rangle$ be an r-RMRA. We construct an r-RA $\mathcal{A}' = \langle Q',\ F',\ \Delta'\rangle$ of type I as follows. We take

$$Q' = Q \times ([r] \xrightarrow{\cong} [r]) \qquad F' = \{(q, f) \in Q' \mid q \in F\}$$

and include in Δ' precisely the transitions given below.[12]

- For each $q \xrightarrow[\mathcal{A}]{R} q'$ and $\pi,\ f : [r] \xrightarrow{\cong} [r]$ and $R_{\alpha\bullet}$ such that $R \parallel (\pi; R_{\alpha\bullet})$, add $(q, f) \xrightarrow[\mathcal{A}']{R_{\alpha'\bullet}} (q', f')$ with $\alpha' = (\pi^{-1}; f)(\alpha)$ and $f' = \pi^{-1}; f$.

[12] Here we view the bijection π as an r-manipulation with component π^0 empty and $\pi^1 = \pi$. Moreover, we write $R \parallel (\pi; R_{\alpha\bullet})$ for the condition "$R \cup (\pi; R_{\alpha\bullet})$ is a valid r-manipulation".

- For each $q \xrightarrow[\mathcal{A}]{i} q'$ and $f : [r] \xrightarrow{\cong} [r]$ add $(q, f) \xrightarrow[\mathcal{A}']{f(i)} (q', f)$.
- For each $q \xrightarrow[\mathcal{A}]{\bullet} q'$ and $f : [r] \xrightarrow{\cong} [r]$ add $(q, f) \xrightarrow[\mathcal{A}']{\bullet} (q', f)$.

The idea is that the second component f of a given state $(q,\ f)$ of $\mathcal{A}'$ maps register locations of configurations in $\mathcal{A}$ in state q to register locations of configurations of $\mathcal{A}'$ in state $(q,\ f)$. For example, the first clause of the construction above says that, if there is an R transition in $\mathcal{A}$ from q to q' and R can be decomposed as some permutation of locations π followed by some refreshes $R_\alpha\bullet$ then, in $\mathcal{A}'$, (q, f) can transition to $(q',\ f')$ on $R_{\alpha'}\bullet$. Since R may refresh those locations in α which are themselves given by permuting locations according to π, it follows that the corresponding register locations are given in $\mathcal{A}'$ by starting from α, undoing π and then consulting f.

Now, let $\mathcal{A}_1$ and $\mathcal{A}_2$ be register automata (of any denomination). We say that a relation $\mathcal{R}$ between configurations of $\mathcal{A}_1$ and $\mathcal{A}_2$ is a *simulation* if, whenever $\kappa_1 \mathcal{R} \kappa_2$,

- if κ_1 is final then so is κ_2;
- if $\kappa_1 \vdash^x_{\mathcal{A}_1} \kappa_1'$ (with $x \in \mathcal{D} \cup \{\varepsilon\}$) then $\kappa_2 \vdash^x_{\mathcal{A}_2} \kappa_2'$ and $\kappa_1' \mathcal{R} \kappa_2'$.

We say that $\mathcal{R}$ is a *bisimulation* if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations.

Lemma 26. *Let $\mathcal{A}$ be an r-RMRA and $\mathcal{A}'$ be the type-I r-RA $\mathcal{A}'$ constructed as above. Then, the relation*

$$\mathcal{R} = \{\, ((q, \tau), (q, f, \hat{\tau})) \mid \tau = f; \hat{\tau} \,\}$$

is a bisimulation[13].

Proof. Note first that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ relate final configurations to final ones. Now, let $(q, \tau)\mathcal{R}(q, f, \hat{\tau})$.

Suppose $(q, \tau) \vdash (q', \tau')$, due to some $q \xrightarrow{R} q'$, so $\tau R \tau'$. We let $\pi : [r] \xrightarrow{\cong} [r]$ be such that, for all i, j, $(i, j) \in \pi$ iff $\tau(i) = \tau'(j)$, and α be an enumeration of the elements of $[r] \setminus \mathbf{ran}(\pi)$. Since $\tau R \tau'$, we have $R \parallel (\pi; R_\alpha\bullet)$. Thus, $(q, f) \xrightarrow[\mathcal{A}']{R_{\alpha'}\bullet} (q', f')$ and so $(q, f, \hat{\tau}) \vdash (q', f', \hat{\tau}')$, for $f' = \pi^{-1}; f$ and $\alpha' = (\pi^{-1}; f)(\alpha)$ and any $\hat{\tau}'$ such that $\hat{\tau} R_{\alpha'}\bullet \hat{\tau}'$. We take $\hat{\tau}' = f^{-1}; \pi; \tau'$ and proceed to show $\hat{\tau} R_{\alpha'}\bullet \hat{\tau}'$. We have:

$$\begin{aligned}
R_{\alpha'}\bullet(i, j) = 1 &\implies i = j \notin (\pi^{-1}; f)(\alpha) \\
&\implies (f^{-1}; \pi)(i) = (f^{-1}; \pi)(j) \notin \alpha \\
&\overset{(1)}{\implies} (\pi; R_\alpha\bullet)(f^{-1}(i), (f^{-1}; \pi)(j)) = 1 \\
&\overset{(2)}{\implies} \tau(f^{-1}(i)) = \tau'((f^{-1}; \pi)(j)) \implies \hat{\tau}(i) = \hat{\tau}'(j)
\end{aligned}$$

where (1) is by definition of $\pi; R_\alpha\bullet$ and (2) is by definition of π and α. Moreover,

$$\begin{aligned}
R_{\alpha'}\bullet(i, j) = 0 &\overset{(3)}{\implies} j \in (\pi^{-1}; f)(\alpha) \implies (f^{-1}; \pi)(j) \in \alpha \\
&\overset{(4)}{\implies} \tau'((f^{-1}; \pi)(j)) \neq \tau(f^{-1}(i)) \implies \hat{\tau}'(j) \neq \hat{\tau}(i)
\end{aligned}$$

[13] Throughout this section we write $(q,\ f,\ \hat{\tau})$ rather than the more cumbersome $((q,\ f),\ \hat{\tau})$.

where (3) is by definition of $R_{\alpha'\bullet}$ and (4) is by definition of α. Now, observing that $f';\hat{\tau}' = \pi^{-1};f;f^{-1};\pi;\tau' = \tau'$, we obtain $(q',\tau')\mathcal{R}(q',f',\hat{\tau}')$.

Conversely, if $(q,f,\hat{\tau}) \vdash (q',f',\hat{\tau}')$, due to some $(q,f) \xrightarrow[\mathcal{A}']{R_{\alpha'\bullet}} (q',f')$ via some $q \xrightarrow[\mathcal{A}]{R} q'$, then $f' = \pi^{-1};f$ and $\alpha' = (\pi^{-1};f)(\alpha)$ and $\hat{\tau}R_{\alpha'\bullet}\hat{\tau}'$, for some π and α such that $R \parallel (\pi;R_{\alpha\bullet})$. Note that $\pi;R_{\alpha\bullet}$ is maximal, in the sense that $R \parallel (\pi;R_{\alpha\bullet})$ implies $R^1 \subseteq (\pi;R_{\alpha\bullet})^1$. Now, $(q,\tau) \vdash (q',\tau')$ for all $\tau R\tau'$. We take $\tau' = \pi^{-1};f;\hat{\tau}'$ and proceed to show $\tau R\tau'$. We have:

$$\begin{aligned} R(i,j) = 1 &\implies (\pi;R_{\alpha\bullet})(i,j) = 1 \implies \pi(i) = j \notin \alpha \\ &\implies f(i) = (\pi^{-1};f)(j) \notin (\pi^{-1};f)(\alpha) = \alpha' \\ &\implies R_{\alpha'\bullet}(f(i),(\pi^{-1};f)(j)) = 1 \\ &\overset{(5)}{\implies} \hat{\tau}(f(i)) = \hat{\tau}'((\pi^{-1};f)(j)) \implies \tau(i) = \tau'(j) \end{aligned}$$

Moreover, if $R(i,j) = 0$ then either $j \in \alpha$ or $\pi(i) \neq j$, and

$$\begin{aligned} (j \in \alpha \vee \pi(i) \neq j) &\implies ((\pi^{-1};f)(j) \in (\pi^{-1};f)(\alpha) \vee (\pi^{-1};f)(j) \neq f(i)) \\ &\overset{(6)}{\implies} \hat{\tau}'(\pi^{-1};f)(j)) \neq \hat{\tau}(f(i)) \implies \tau'(j) \neq \tau(i) \end{aligned}$$

where (5) and (6) follow from $\hat{\tau}R_{\alpha'\bullet}\hat{\tau}'$. Now, observing that $f';\hat{\tau}' = \pi^{-1};f;\hat{\tau}' = \tau'$, we obtain $(q',\tau')\mathcal{R}(q',f',\hat{\tau}')$.

The cases of the other transitions are straightforward. □

Now, let $\mathcal{A} = \langle Q, F, \Delta\rangle$ be an r-RA of type I. We construct a $2r$-RA $\mathcal{A}' = \langle Q', F, \Delta'\rangle$ of type II accepting the same languages as $\mathcal{A}$ by simply decomposing the $R_{\alpha\bullet}$-labelled transitions of $\mathcal{A}$ into constituent $R_{i\bullet}$-labelled ones. Note that, because of possible name-reuses in the composition $R_{i_1\bullet};\cdots;R_{i_m\bullet}$, the latter is not equivalent to $R_{\{i_1,\cdots,i_m\}\bullet}$. To avoid unintentional name-reuse, we add an additional r registers to record old names that should not be reused in a transition. In particular, we take

$$Q' = (Q \times ([r] \overset{\cong}{\to} [2r])) \cup \hat{Q} \qquad F' = \{(q,f) \in Q' \mid q \in F\}$$

where $\hat{Q} = \{\,(t,k,f) \in (\Delta \times [r] \times \Phi) \mid t = q \xrightarrow[\mathcal{A}]{R_{\{i_1,\cdots,i_m\}\bullet}} q',\ k < m\,\}$ is a set of auxiliary states that will be used for breaking down transitions of the form $q \xrightarrow[\mathcal{A}]{R_{\alpha\bullet}} q'$. Here $[r] \overset{\cong}{\to} [2r]$ denotes the set of (total) injections from $[r]$ to $[2r]$. We let $\Phi = \{\, f \in [2r] \overset{\cong}{\rightharpoonup} [2r] \mid \mathbf{dom}(f) = [r+i],\ 0 \leq i < r\,\}$ be re-indexing functions that allow us to locate the r registers of $\mathcal{A}$ inside the $2r$ registers of $\mathcal{A}'$ (i.e. the i-th $\mathcal{A}$-register is to be found in the $f(i)$th $\mathcal{A}'$-register), but also specify the names that should not be reused at this point (all names in $\mathcal{A}'$-registers $\mathbf{ran}(f) \setminus f([r])$). Then, the transitions of Δ' are obtained by dividing transitions of the form $q \xrightarrow[\mathcal{A}]{R_{\{i_1,\cdots,i_m\}\bullet}} q'$ into m steps, where at each step j we stipulate that a fresh name should be introduced in register i_j, with freshness specified with respect to the range of the re-indexing function f stored in the current state of $\mathcal{A}'$. These re-indexing functions need to be updated along the way. Moreover, sometimes the fresh names may actually be already resident in the part of the $2r$ registers that is not covered by f.

Lemma 27. *Let $\mathcal{A}$ be a type-I r-RA and $\mathcal{A}'$ be the type-II $2r$-RA constructed as above. For all $\mathcal{A}$-configurations (q,τ), $\mathcal{L}(\mathcal{A})(q,\tau) = \mathcal{L}(\mathcal{A}')(q,\{\, i \mapsto i \mid i \in [r]\,\},\tau)$.*

Proof. We here give the full definition of Δ'. Note first that, by convention, given transition $t = q \xrightarrow[\mathcal{A}]{R_{\{i_1,\cdots,i_m\}}} q'$ and $f : [r] \xrightarrow{\cong} [2r]$, we write $(t,0,f)$ for (q,f). Given $f \in \Phi$, we let $f_\ominus$ be the least number in $[2r] \setminus \mathbf{dom}(f)$, and $f_\oplus$ to be the least number in $[2r] \setminus \mathbf{ran}(f)$. We build Δ' as follows.

- For each $t = q \xrightarrow[\mathcal{A}]{R_{\{i_1,\cdots,i_m\}^\bullet}} q'$ and $(t,k,f) \in Q'$ with $0 \leq k < m-1$ and $|f| \leq r+k$, add:
  - $(t,k,f) \xrightarrow[\mathcal{A}']{R_{f_\oplus^\bullet}} (t,k{+}1,f')$ with $f' = f[f_\ominus \mapsto f(i_k),\ i_k \mapsto f_\oplus]$, and
  - for each $j \in [2r] \setminus \mathbf{ran}(f)$, $(t,k,f) \xrightarrow[\mathcal{A}']{\varepsilon} (t,k{+}1,f[f_\ominus \mapsto f(i_k), i_k \mapsto j])$,
- For each $t = q \xrightarrow[\mathcal{A}]{R_{\{i_1,\cdots,i_m\}^\bullet}} q'$, $m > 0$, and $(t,m-1,f) \in Q'$, add:
  - $(t,m-1,f) \xrightarrow[\mathcal{A}']{R_{f_\oplus^\bullet}} (q', f' \restriction [r])$ with $f' = f[i_m \mapsto f_\oplus]$, and
  - for each $j \in [2r] \setminus \mathbf{ran}(f)$, add $(t,m-1,f) \xrightarrow[\mathcal{A}']{\varepsilon} (q', f[i_m \mapsto j] \restriction [r])$,
- For each $q \xrightarrow[\mathcal{A}]{R_\emptyset^\bullet} q'$, add $(q,f) \xrightarrow[\mathcal{A}']{\varepsilon} (q,f)$.
- For each $q \xrightarrow[\mathcal{A}]{i} q'$ and f, add $(q,f) \xrightarrow[\mathcal{A}']{f(i)} (q',f)$.
- For each $q \xrightarrow[\mathcal{A}]{\bullet} q'$ and f, add $(q,f) \xrightarrow[\mathcal{A}']{\bullet} (q',f)$ and, for all $j \in [2r] \setminus \mathbf{ran}(f)$, add $(q,f) \xrightarrow[\mathcal{A}']{j} (q',f)$.

Now, according to the argument given above, we can see that, for each transition $q \xrightarrow[\mathcal{A}]{R_a^\bullet} q'$, $f : [r] \xrightarrow{\cong} [2r]$ and pair of r- and $2r$-assignments $\tau, \hat{\tau}$ such that $\tau = f;\hat{\tau}$, if $(q,\tau) \vdash (q',\tau')$ then $(q,f,\hat{\tau}) \vdash^* (q',f',\hat{\tau}')$, for $\hat{\tau}', f'$ such that $\tau' = f';\hat{\tau}'$. Also, conversely, if $(q,f,\hat{\tau}) \vdash^* (q',f',\hat{\tau}')$ then $(q,\tau) \vdash (q',f;\hat{\tau}')$. Our statement then follows from these two observations. □

Finally, let $\mathcal{A} = \langle Q, F, \Delta\rangle$ be an r-RA of type II. We construct a type-III $(r{+}1)$-RA $\mathcal{A}' = \langle Q', F', \Delta'\rangle$ as follows. We take

$$Q' = Q \times ([r] \xrightarrow{\cong} [r+1]) \qquad F' = \{(q,f) \in Q' \mid q \in F\}$$

and include in Δ' precisely the transitions given below. Here, the idea is to simulate the $\xrightarrow[\mathcal{A}]{\bullet}$ transition by using the extra $(r{+}1)$th register to generate a locally fresh element of the infinite alphabet and then read exactly that element from the input word. Given $f : [r] \xrightarrow{\cong} [r+1]$ we write $f_\oplus$ for the unique element of the set $[r+1] \setminus \mathbf{ran}(f)$. For all $q \in Q$ and $f : [r] \xrightarrow{\cong} [r+1]$:

- For each $q \xrightarrow[\mathcal{A}]{R_i^\bullet} q'$ add $(q,f) \xrightarrow[\mathcal{A}']{R_{f(i)}^\bullet} (q,f)$ and $(q,f) \xrightarrow[\mathcal{A}']{\varepsilon} (q, f[i \mapsto f_\oplus])$.
- For each $q \xrightarrow[\mathcal{A}]{i} q'$ add $(q,f) \xrightarrow[\mathcal{A}']{f(i)} (q',f)$.

- For each $q \xrightarrow[\mathcal{A}]{\bullet} q'$ add $(q,f) \xrightarrow[\mathcal{A}']{R^+_{f_\oplus}\bullet} (q',f)$ and $(q,f) \xrightarrow[\mathcal{A}']{f_\oplus} (q',f)$.
- For each $q \xrightarrow[\mathcal{A}]{\varepsilon} q'$ add $(q,f) \xrightarrow[\mathcal{A}']{\varepsilon} (q',f)$.

Lemma 28. *Let $\mathcal{A}$ be a type-II r-RA and $\mathcal{A}'$ be the type-III $(r{+}1)$-RA constructed as above. Then, the relation*

$$\mathcal{R} = \{\, ((q,\tau),(q,f,\hat{\tau})) \mid \tau = f;\hat{\tau} \,\}$$

is a bisimulation.

Proof. Note first that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ relate final configurations to final ones. Now let $(q,\tau)\mathcal{R}(q,f,\hat{\tau})$.

Suppose $(q,\tau) \vdash (q',\tau')$, due to some $q \xrightarrow{R_i\bullet} q'$, so $\tau' = \tau[i \mapsto a]$ for some fresh $a \in \mathcal{D}$. If $a = \hat{\tau}(f_\oplus)$ then, from $(q,f) \xrightarrow[\mathcal{A}']{\varepsilon} (q, f[i \mapsto f_\oplus])$, we have $(q,f,\hat{\tau}) \vdash (q,f',\hat{\tau})$ with $f' = f[i \mapsto f_\oplus]$. Moreover, since $\tau = f;\hat{\tau}$, we also have $\tau' = f';\hat{\tau}$, so $(q',\tau')\mathcal{R}(q',f',\hat{\tau})$. If $a \neq \hat{\tau}(f_\oplus)$ then $\tau = f;\hat{\tau}$ implies that a is fresh for $\hat{\tau}$. Thus, $(q,f) \xrightarrow[\mathcal{A}']{R_{f(i)}\bullet} (q',f)$ implies $(q,f,\hat{\tau}) \vdash (q',f,\hat{\tau}')$ with $\hat{\tau}' = \hat{\tau}[f(i) \mapsto a]$. Moreover, $\tau' = f;\hat{\tau}'$ so $(q',\tau')\mathcal{R}(q',f,\hat{\tau}')$.
Conversely, suppose $(q,f,\hat{\tau}) \vdash (q',f',\hat{\tau}')$, due to $(q,f) \xrightarrow[\mathcal{A}']{R_{f(i)}\bullet} (q',f)$, itself due to some $q \xrightarrow[\mathcal{A}]{R_i\bullet} q'$. Then, $f' = f$ and $\hat{\tau}' = \hat{\tau}[f(i) \mapsto a]$, some $a \notin \mathbf{ran}(\hat{\tau})$, and we can see that $(q,\tau) \vdash (q',\tau')$ with $\tau' = \tau[i \mapsto a]$ and $(q',\tau')\mathcal{R}(q',f,\hat{\tau}')$. If the transition is due to $(q,f) \xrightarrow[\mathcal{A}']{\varepsilon} (q', f[i \mapsto f_\oplus])$ then, since $\hat{\tau}(f_\oplus) \notin \mathbf{ran}(\tau)$, we have $(q,\tau) \vdash (q',\tau')$ with $\tau' = \tau[i \mapsto \hat{\tau}(f_\oplus)]$ and $(q',\tau')\mathcal{R}(q',f',\hat{\tau})$.

Suppose $(q,\tau) \vdash^a (q',\tau)$, due to $q \xrightarrow[\mathcal{A}]{\bullet} q'$. Then, if $a = \hat{\tau}(f_\oplus)$, from $(q,f) \xrightarrow[\mathcal{A}]{f_\oplus} (q',f)$ we obtain $(q,f,\hat{\tau}) \vdash^a (q',f,\hat{\tau})$ and, since $\tau = \tau'$, $(q',\tau')\mathcal{R}(q',f,\hat{\tau})$. On the other hand, if $a \notin \mathbf{ran}(\hat{\tau})$, we use the transition $(q',f) \xrightarrow[\mathcal{A}']{R^+_{f_\oplus}\bullet} (q',f)$ to obtain $(q,f,\hat{\tau}) \vdash^a (q',f,\hat{\tau}')$, with $\hat{\tau}' = \hat{\tau}[f_\oplus \mapsto a]$, and we can see that $(q',\tau)\mathcal{R}(q',f,\hat{\tau}')$. Also, using essentially the same argument, we can show that if $(q,f,\hat{\tau}) \vdash^a (q',f,\hat{\tau}')$, with a fresh or $a = \hat{\tau}(f_\oplus)$, then $(q,\tau) \vdash^a (q',\tau)$ with $(q',\tau)\mathcal{R}(q',f,\hat{\tau}')$.

The cases of the other transitions are straightforward. □

Summing up, we have the following.

Theorem 29. *Let $\mathcal{A}$ be an r-RMRA. We can construct a $(2r{+}1)$-RA$_{nr}$ $\hat{\mathcal{A}}$, of size $O(2^{p(|\mathcal{A}|)})$ for some polynomial p, such that, for each $\mathcal{A}$-configuration κ there is a $\hat{\mathcal{A}}$-configuration $\hat{\kappa}$ such that $\mathcal{L}(\mathcal{A})(\kappa) = \mathcal{L}(\hat{\mathcal{A}})(\hat{\kappa})$.*

Proof. By consecutively applying the three previous lemmas, and using the fact that bisimilarity implies language equivalence, we obtain the statement for $\hat{\mathcal{A}}$ a type-III $(2r{+}1)$-RA of exponential size. From the latter we obtain a $(2r{+}1)$-RA$_{nr}$ by simply breaking each transition of the form $q \xrightarrow[\hat{\mathcal{A}}]{R^+_i\bullet} q'$ into two transitions: $q \xrightarrow{R_i\bullet} \cdot \xrightarrow{i} q'$. Epsilon transitions can be removed using the standard procedure of computing epsilon closures for each state. □

References

[1] R. Alur, P. Cerný, S. Weinstein, Algorithmic analysis of array-accessing programs, ACM Trans. Comput. Log. 13 (3) (2012) 27.

[2] M. F. Atig, A. Bouajjani, S. Qadeer, Context-bounded analysis for concurrent programs with dynamic creation of threads, Log. Methods in Comput. Sci. 7 (4).

[3] H. Björklund, T. Schwentick, On notions of regularity for data languages, Theor. Comput. Sci. 411 (4-5) (2010) 702–715.

[4] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin, Two-variable logic on data words, ACM Trans. Comput. Log. 12 (4) (2011) 27.

[5] A. Bouajjani, J. Esparza, O. Maler, Reachability analysis of pushdown automata: Application to model-checking, in: Proceedings of 8th International Conference on Concurrency Theory, CONCUR, vol. 1243 of Lecture Notes in Computer Science, Springer, 1997, pp. 135–150.

[6] A. Bouajjani, S. Fratani, S. Qadeer, Context-bounded analysis of multithreaded programs with dynamic linked structures, in: Proceedings of the 19th International Conference on Computer Aided Verification , CAV, vol. 4590 of Lecture Notes in Computer Science, Springer, 2007, pp. 207–220.

[7] A. Bouajjani, P. Habermehl, R. Mayr, Automatic verification of recursive procedures with one integer parameter, Theor. Comput. Sci. 295 (2003) 85–106.

[8] E. Y. C. Cheng, M. Kaminski, Context-free languages over infinite alphabets, Acta Inf. 35 (3) (1998) 245–267.

[9] S. A. Cook, Characterizations of pushdown machines in terms of time-bounded computers, J. of the Assoc. for Comput. Mach. 18(1) (1971) 4–18.

[10] S. Demri, R. Lazić, LTL with the freeze quantifier and register automata, ACM Trans. Comput. Log. 10 (3) (2009) 16:1–16:30.

[11] M. J. Gabbay, A. M. Pitts, A new approach to abstract syntax with variable binding, Form. Asp. of Comput. 13 (2002) 341–363.

[12] R. Grigore, D. Distefano, R. L. Petersen, N. Tzevelekos, Runtime verification based on register automata, in: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2013, vol. 7795 of Lecture Notes in Computer Science, Springer, 2013, pp. 260–276.

[13] M. Kaminski, N. Francez, Finite-memory automata, Theor. Comput. Sci. 134 (2) (1994) 329–363.

[14] M. Kaminski, D. Zeitlin, Finite-memory automata with non-deterministic reassignment, Int. J. Found. Comput. Sci. 21 (5) (2010) 741–760.

[15] A. N. Maslov, Multilevel stack automata, Probl. of Inf. Transm. 12 (1976) 3843.

[16] A. S. Murawski, S. J. Ramsay, N. Tzevelekos, A contextual equivalence checker for IMJ*, in: Proceedings of the 13th International Symposium on Automated Technology for Verification and Analysis, ATVA, vol. 9364 of Lecture Notes in Computer Science, Springer, 2015, pp. 234–240.

[17] A. S. Murawski, S. J. Ramsay, N. Tzevelekos, Game semantic analysis of equivalence in IMJ, in: Proceedings of the 13th International Symposium on Automated Technology for Verification and Analysis, ATVA, vol. 9364 of Lecture Notes in Computer Science, Springer, 2015, pp. 411–428.

[18] A. S. Murawski, N. Tzevelekos, Algorithmic nominal game semantics, in: Proceedings of 20th European Symposium on Programming, ESOP, vol. 6602 of Lecture Notes in Computer Science, Springer-Verlag, 2011, pp. 419–438.

[19] A. S. Murawski, N. Tzevelekos, Algorithmic games for full ground references, in: Proceedings of 39th International Colloquim on Automata, Languages and Programming ICALP, vol. 7392 of Lecture Notes in Computer Science, Springer, 2012, pp. 312–324.

[20] F. Neven, T. Schwentick, V. Vianu, Finite state machines for strings over infinite alphabets, ACM Trans. Comput. Log. 5 (3) (2004) 403–435.

[21] H. Sakamoto, D. Ikeda, Intractability of decision problems for finite-memory automata, Theor. Comput. Sci. 231 (2) (2000) 297–308.

[22] L. Segoufin, Automata and logics for words and trees over an infinite alphabet, in: Proceedings of 20th International Workshop on Computer Science Logic, CSL, vol. 4207 of Lecture Notes in Computer Science, Springer, 2006, pp. 41–57.

[23] T. Tan, On pebble automata for data languages with decidable emptiness problem, J. Comput. Syst. Sci. 76 (8) (2010) 778–791.