Reachability in Pushdown Register Automata*

Andrzej S. Murawski[1], Steven J. Ramsay[1], and Nikos Tzevelekos[2]

[1] University of Warwick
[2] Queen Mary University of London

Abstract. We investigate reachability in pushdown automata over infinite alphabets: machines with finite control, a finite collection of registers and pushdown stack. First we show that, despite the stack's unbounded storage capacity, in terms of reachability/emptiness these machines can be faithfully represented by using only $3r$ elements of the infinite alphabet, where r is the number of registers. Moreover, this bound is tight. Next we settle the complexity of the associated reachability/emptiness problems. In contrast to register automata, where differences in register storage policies gave rise to differing complexity bounds, the emptiness problem for pushdown register automata is EXPTIME-complete in all cases. We also provide a solution to the global reachability problem, based on representing pushdown configurations with a special register automaton. Finally, we examine extensions of pushdown storage to higher orders and show that reachability is undecidable already at order 2, unlike in the finite alphabet case.

1 Introduction

Recent years have seen lively interest in automata over infinite alphabets, driven by applications in quite diverse areas where abstraction by a finite domain was deemed unsatisfactory. A case in point are markup languages [18,4], most notably XML, which permit the use of potentially unbounded data values in documents and allow queries to perform comparison tests on such data. A similar scenario occurs in reference-based programming languages, such as object-oriented [6,2,12] or ML-like languages [16,17], where memory is managed with the help of abstract addresses (reference names) that can be created afresh, compared for equality but little else. Other examples include array-accessing programs [1] as well as programs with restricted integer parameters [7].

Such applications call for a robust theory of automata over infinite alphabets, which will match our understanding of the finite-alphabet setting. Thus the limits will be exposed and a complexity-theoretic guide established for applications. A lot of the groundwork, surveyed in [20,3], was already dedicated to uncovering a notion of "regularity" in the infinite-alphabet case. One way to extend the concept of finite memory to such a setting consists of introducing a fixed number of *registers* for storing elements of the alphabet [13]. Another strand of

* Research supported by the Engineering and Physical Sciences Research Council (EP/J019577/1) and the Royal Academy of Engineering (RF: Tzevelekos).

work aimed to identify the infinite-alphabet "context-free" languages. Cheng and Kaminski [8] introduced context-free grammars over infinite alphabets and defined a corresponding notion of pushdown automata. Segoufin presents a similar definition in [20], albeit couched in a way suitable to process data words.

Our paper is devoted to studying exactly such computational scenarios through a study 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], and in particular their application to game-semantics-based verification [17]. We present several new results on the complexity of reachability testing. Altogether they 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. Even though pushdown register systems have no bound on the number of elements of the alphabet that can be stored at any instant, we show that, over the course of a run, they can nevertheless 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 involving 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 problems, and language emptiness. 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. In contrast, in the pushdown case, 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, while it is EXPTIME-hard already in the case where only distinct elements are allowed. In the last 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).

We show how to conduct *global* reachability analysis, which asks for a representation of all configurations from which a specified set of 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 using a variant of register automata.

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.

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 ith register; – $push(i)$ pushes the symbol currently in the ith register on the stack; – $pop(i)$ pops the stack if the top symbol is the same as that stored in the ith 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.*

[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 $pop(1)$.

- $op = i^\bullet$, $\forall j.\, \tau_2(i) \neq \tau_1(j)$, $\forall j \neq i.\, \tau_2(j) = \tau_1(j)$ *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* $\kappa_0 = (q_I, \tau_I, \epsilon)$.

Remark 3. 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], [20] 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 [20], 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 [20] 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 ith 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 [13,8,18], 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 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 4 (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 5 (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. For $n \leq 1$ the result is trivial. Otherwise, the difficult case arises when 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: $\rho_1 = (q_0, \tau'_0, \epsilon) \vdash^k (q_k, \tau'_k, \epsilon)$ and $\rho_2 = (q_k, \tau'_k, \epsilon) \vdash^{n-k} (q_n, \tau'_n, \epsilon)$ 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 4): $(q_0, \pi_i \cdot \tau_0, \epsilon) \vdash^k (q_k, \pi_i \cdot \tau_k = \pi_j \cdot \tau_k, \epsilon) \vdash^{n-k} (q_n, \pi_j \cdot \tau_n, \epsilon)$ 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 6. *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 7 (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, 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 8. The $3r$ bound given above can be adapted to the automata presentations of [8,20] yielding bounds $3r + \Theta(1)$. An adaptation of Lemma 7 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 9).

4 Reachability is EXPTIME-complete

We consider the following decision problem, call it 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.

Theorem 9. *r-PDRS Reach and language emptiness for infinite-alphabet pushdown automata [8] and register pushdown automata [20] are solvable in exponential time.*

Proof. Lemma 5 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 3, 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 [20] 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 5 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. □

The bound given above is tight: we simulate a polynomial-space Turing machine with a stack (aka polynomial-space auxiliary pushdown automaton [9]), which has an EXPTIME-complete halting problem[5].

Theorem 10. *r-PDRS Reach is EXPTIME-hard.*

Proof (sketch). For simplicity, let us assume a binary tape alphabet. The main challenge in the proof is the modelling of n tape cells using $p(n)$ registers, for a polynomial p. Recall that register assigments are injective, so it is not clear which registers represent 0's and which represent 1's. Thus, to encode n bits $b_1, \cdots, b_n$, we shall use a special encoding scheme based on $2n$ names $r_1, \cdots, r_{2n} \in \mathcal{D}$ stored in registers and an auxiliary "mask" of names $m_1, \cdots, m_{2n} \in \mathcal{D}$ stored on the stack. The registers and masks will be related by $\{r_{2j-1}, r_{2j}\} = \{m_{2j-1}, m_{2j}\}$ and $b_j = 0$ will be represented by the case $r_{2j-1} = m_{2j-1}, r_{2j} = m_{2j}$. Note that, due to injectivity, both r_j's and m_j's cannot be present in registers at the same time and hence the latter will be pushed on the stack. However, the stack is also needed for pushing and popping ordinary stack symbols by the Turing machine, so masks will not always be at the top of stack at the time when they are needed for decoding[6]. We overcome this obstacle by employing 3 different masks for encoding memory: one is used whilst simulating push-transitions (*push-mode*), one for pops (*pop-mode*) and an auxiliary one to ensure continuity between the different instances of masks. Let us call these masks M_1, M_2 and M_3 respectively.

In push-mode, instead of popping M_1 from the stack in order to compare it with the registers and hence decode the memory, we will be *guessing* it and pushing the guess onto the stack, on the understanding that the correctness of each guess (call it $\hat{M}_1$) is to be verified later in the corresponding pop steps. Moreover, in push-mode we will also be pushing the mask M_2 so that it is

[4] Through a careful reading of the argument for emptiness in [8] one can infer an exponential upper bound, but here Lemma 5 gives a direct argument.

[5] 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.

[6] For example, after simulating a push-transition, the mask used for realising the transition will be hidden by the pushed symbol and thus unavailable to support the next transition.

readily available for pop-mode. When it is time to switch to pop-mode, the tape content so far encoded with mask M_1 will be re-encoded with M_2 so that the forthcoming pop-move can be simulated with M_2. During pop-transitions, in addition to stack symbols and the mask M_2 used for decoding, we will also pop the accompanying guessed mask $\hat{M}_1$ and verify its correctness by comparing it with the last unverified $\hat{M}_1$, which is stored in registers apart from the simulated memory. Because at the bottom of the stack we have the actual mask M_1, such equality comparisons will eventually assert that $\hat{M}_1 = M_1$ for all guesses $\hat{M}_1$.

A final complication arises when we want to switch from pop-mode to push-mode. We said that, when popping, we verify the guesses $\hat{M}_1$. Thus, if a push follows a pop, the mask $\hat{M}_1$ that resides in the registers needs to be pushed back on the stack so that it can be verified later once we return to pop-mode. At the same time, we need to store in our registers some content X, so that X and $\hat{M}_1$ encode the current tape content. However, the formation of X destroys $\hat{M}_1$ in the registers. To prevent the information from being lost, we make another guess $\hat{\hat{M}}_1$ and use the third mask M_3 to check that the guess was correct (more precisely, on the stack we store M_3 and some $\hat{M}_3$ such that $\hat{M}_1 = \hat{\hat{M}}_1$ iff $\hat{M}_3 = M_3$). Whether $\hat{M}_3 = M_3$ holds is verified in a later pop step. □

The EXPTIME-hardness carries over to the language emptiness problem associated with infinite-alphabet pushdown automata [8] and register pushdown automata [20]. 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 [19][7] (injective assignment) vs PSPACE-completeness (non-injective assignment) [10]. Perhaps surprisingly, we have shown the presence of pushdown store cushions the differences and there is no gap analogous to that between [8] and [20].

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 $\mathcal{C}$ of a set of configurations of $\mathcal{S}$, one can construct, in exponential time, a representation of the set of configurations $\mathsf{Pre}^*_{\mathcal{P}}(\mathcal{C})$ from which $\mathcal{S}$ can reach a configuration in $\mathcal{C}$. 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***

[7] 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.

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

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

Definition 11. *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 $\mathsf{OP}_r = [r] \cup \{\bullet\} \cup \mathsf{RegMan}_r$.*

The operations of RMRAs generalise the stack-free operations of PDRSs: $i \in [r]$ specifies reading a name already present in the ith 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 $(q,\,\tau)$ may transition to any configuration $(q,\,\tau')$ 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 12. *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 $(p,\,\tau,\,s)$ whenever $s \in \mathcal{L}(\mathcal{A})(p,\,\tau)$. We write $\mathcal{C}(\mathcal{A})$ 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 $\mathcal{C}(\mathcal{A})$, i.e. we aim to construct a representation of $\mathsf{Pre}^*_{\mathcal{P}}(\mathcal{C}(\mathcal{A}))$.

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 $i = j$ (respectively $i^\bullet$) just if $R(i,\,j) \neq 0$ and $[i \in \mathbf{dom}\,R^1 \vee j \in \mathbf{ran}\,R^1]$ implies $R^1(i) = j$ (resp. $i \notin \mathbf{dom}\,R^1$) and in that case we write $R \,||\, i = j$ (resp. $R \,||\, i^\bullet$). So, the meaning of $R \,||\, i^\bullet$, is that i in the situation before R may be locally fresh with respect to the situation after R. If $R \,||\, i = j$ (resp. $R \,||\, i^\bullet$) 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 13. *Given an r-PDRS $\mathcal{S}$ over states P and an r-RMRA $\mathcal{A}$ over states Q and transitions Δ and 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 $R \mid\mid i = j$ in rule (ii), and $R \mid\mid i^\bullet$ 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 14. *Given r-PDRS $\mathcal{S}$ and r-RMRA $\mathcal{A}$ as above, $\mathcal{C}(\mathsf{SAT}(\mathcal{A})) = \mathsf{Pre}^*_{\mathcal{P}}(\mathcal{C}(\mathcal{A}))$.*

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 remark 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 $\mathsf{OP}_r^- = [r] + \{ R_{i\bullet} \mid i \in [r] \}$.

Lemma 15. *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 state 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*:

- $push_1^a\langle a_l, \ldots, a_1\rangle = \langle a, a_l \ldots, a_1\rangle$ 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 16. *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:*

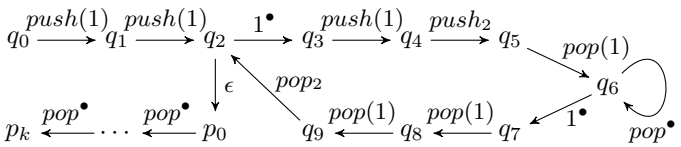
$$Op_r^n = Op_r \cup \{\, push_k,\, pop_k \mid 2 \leq k \leq n \,\}$$

A configuration of an r-nPDRS is a triple (q, τ, 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, τ_2, s_2) is said to be a successor of a configuration (q_1, τ_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 17. 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 state reachability problem for higher-order PDRS, we reduce from the emptiness problem for weak pebble automata, which is known to be undecidable [18,21]. We find it convenient to use pebbles, as the push and pop instructions have a direct analogue in placing and lifting a pebble.

Theorem 18. *The state-reachability problem for r-nPDRS is undecidable for any $n > 0$.*

Proof (Sketch). Given a weak k-PA $\mathcal{A}$, we construct a 1-2PDRS $\mathcal{S}$ that first guesses a word w and then checks that $w \in \mathcal{L}(\mathcal{A})$ by simulating an accepting run of $\mathcal{A}$ on w. To simulate $\mathcal{A}$ running on input w in some state q with head pebble $m \leq k$ and where each of its placed pebbles $i \in \{1, \ldots, m\}$ is over some position $p(i)$ of w, the construction has $\mathcal{S}$ in a configuration of shape: $((q, m), d, \langle\langle c_m, \ldots, c_1, a_n, b_n, \ldots, a_1, b_1, \#, \#\rangle\rangle)$. The state component (q, m) of the configuration records both the state and the index of the head pebble of $\mathcal{A}$; d is the data value stored in the single register. The stack component, which is a 2-stack containing a single 1-stack, records the input word and positions of the pebbles. The input w is encoded by the indexed word $(a_n, b_n) \cdots (a_1, b_1)$, in which $w = b_n \cdots b_1$ and, for all $i, j \in [n]$, $a_i \neq b_j$ and, when $i \neq j$, $a_i \neq a_j$ (such distinctness can be guaranteed by using the technique of Example 17). The positions of the pebbles are encoded by the vector $c_m \cdots c_1$, with $c_i = a_j$ iff $p(i) = n - j + 1$. The pair ##, with # different to all other elements, marks the bottom of the stack.

Under this encoding scheme, the placement of a new pebble is simply copying the top element of the 1-stack and the lifting of the head pebble is simply popping the top element of the 1-stack. To move the head pebble right and, more generally, to check the applicability of a given transition of $\mathcal{A}$, requires interrogating the data structure held on the stack. However, since all the relevant data from the simulation is encoded into the state of $\mathcal{S}$ and its single 1-stack (no relevant data is encoded in the register), by using $push_2$ and pop_2 this data can be preserved, interrogated (which will likely result in elements being discarded) and then restored without any overall loss of information.

For example, to simulate moving the head pebble m right, $\mathcal{S}$ first pops the top of its 1-stack, c_m (encoding $p(m)$), into its register. It then takes a copy

of the whole 1-stack using $push_2$, thus preserving a snapshot of the simulation. In the working copy it discards the vector $c_m \cdots c_1$ and then loops, repeatedly using $pop^\bullet$ to discard each element of the indexed word until it finds (a_i, b_i), the unique pair such that $a_i = c_m$, the value in its register (*discarding* of the top of stack can be arranged by allowing a transition to be made on either of $pop(1)$ or $pop^\bullet$). Finally it discards a_i and b_i and replaces the contents of its register by the new top of 1-stack, a_{i-1}, which encodes the position one place to the right of the head pebble. The working 1-stack can then be discarded using pop_2 and new head position, a_{i-1}, pushed onto the restored 1-stack. Checking applicability of transitions is similar. □

References

1. R. Alur, P. Cerný and S. Weinstein. Algorithmic analysis of array-accessing programs. *ACM Trans. Comput. Log.*, 13(3), 2012.
2. M. F. Atig, A. Bouajjani and S. Qadeer. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. *Log. Meth. Comput. Sci.*, 7(4), 2011.
3. H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5), 2010.
4. M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4), 2011.
5. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, 1997.
6. A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV*, 2007.
7. A. Bouajjani, P. Habermehl and R. Mayr. Automatic verification of recursive procedures with one integer parameter. *Theor. Comput. Sci.*, 295: 85-106, 2003.
8. E. Y. C. Cheng and M. Kaminski. Context-free languages over infinite alphabets. *Acta Inf.*, 35(3):245–267, 1998.
9. S. A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM*, 18(1):4–18, 1971.
10. S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3), 2009.
11. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13, 2002.
12. R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In *TACAS*, 2013.
13. M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2), 1994.
14. M. Kaminski and D. Zeitlin. Finite-memory automata with non-deterministic reassignment. *Int. J. Found. Comput. Sci.*, 21(5), 2010.
15. A. N. Maslov. Multilevel stack automata. *Probl. of Inf. Transm.*, 12, 1976.
16. A. S. Murawski and N. Tzevelekos. Algorithmic nominal game semantics. In *ESOP*, 2011.
17. A. S. Murawski and N. Tzevelekos. Algorithmic games for full ground references. In *ICALP*, 2012.
18. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3), 2004.
19. H. Sakamoto and D. Ikeda. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.*, 231(2), 2000.
20. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, 2006.
21. T. Tan. On pebble automata for data languages with decidable emptiness problem. *J. Comput. Syst. Sci.*, 76(8), 2010.