A Saturation Method for Collapsible Pushdown Systems*

C. Broadbent[1], A. Carayol[2], M. Hague[1,2], and O. Serre[1]

[1] LIAFA, Université Paris Diderot – Paris 7 & CNRS
[2] LIGM, Université Paris-Est & CNRS

Abstract. We introduce a natural extension of collapsible pushdown systems called annotated pushdown systems that replaces collapse links with stack annotations. We believe this new model has many advantages. We present a saturation method for global backwards reachability analysis of these models that can also be used to analyse collapsible pushdown systems. Beginning with an automaton representing a set of configurations, we build an automaton accepting all configurations that can reach this set. We also improve upon previous saturation techniques for higher-order pushdown systems by significantly reducing the size of the automaton constructed and simplifying the algorithm and proofs.

1 Introduction

Via languages such as C++, Haskell, Javascript, Python, or Scala, modern day programming increasingly embraces higher-order procedure calls. This is a challenge for software verification, which usually does not model recursion accurately, or models only first-order calls (e.g. SLAM [2] and Moped [29]). Collapsible pushdown systems (collapsible PDS) are an automaton model of (higher-order recursion) schemes [11, 24], which allow reasoning about higher-order recursion.

Collapsible pushdown systems are a generalisation of higher-order pushdown systems (higher-order PDS). Higher-order PDS provide a model of schemes subject to a technical constraint called *safety* [23, 19] and are closely related to the Caucal hierarchy [9]. These systems extend the stack of a pushdown system to allow a nested "stack-of-stacks" structure. Recently it has been shown by Parys that safety is a genuine constraint on definable traces [26]. Hence, to model higher-order recursion fully, we require collapsible PDS, which — using an idea from *panic automata* [20] — add additional *collapse* links to the stack structure. These links allow the automaton to return to the context in which a character was added to the stack.

These formalisms are known to have good model-checking properties. For example, it is decidable whether a given μ-calculus formula holds on the execution graph of a scheme [24] (or collapsible PDS [14]). Although, the complexity of

* This work was supported by the Fondation Sciences Mathématiques de Paris and by the following projects: AMIS (ANR 2010 JCJC 0203 01 AMIS), FREC (ANR 2010 BLAN 0202 02 FREC) and VAPF (Région Île de France).

such analyses is high — for an order-n collapsible PDS, reachability checking is complete for $(n-1)$-EXPTIME, while μ-calculus is complete for n-EXPTIME — the problem becomes PTIME if the arity of the recursion scheme, and the number of alternations in the formula, is bounded. The same holds true for collapsible PDS when the number of control states is bounded. Furthermore, when translating from a scheme to a collapsible PDS, it is the arity that determines the number of control states [14]. It has been shown by Kobayashi [21] that these analyses can be performed in practice. For example, resource usage properties of programs of orders up to five can be verified in a matter of seconds.

Kobayashi *et al.*'s approach uses *intersection types*. In the order-1 case, an alternative approach called *saturation* has been successfully implemented by tools such as Moped [29] and PDSolver [16]. Saturation techniques begin with a small automaton — representing a set of configurations — and add new transitions as they become necessary until a fixed point is reached. These algorithms, then, naturally do not pay the worst case complexity immediately, and hence, represent ideal algorithms for efficient verification. Furthermore, they also provide a solution to the *global* model checking problem: that is, determining the set of all system states that satisfy a property. This is particularly useful when, for example, composing analyses. Furthermore, when testing reachability from a given initial state, we may terminate the analysis as soon as this state is found. That is, we do not need to compute the whole fixed point.

Our first contribution is a new model of higher-order execution called *annotated pushdown systems* (annotated PDS)[1], which replace the collapse links of a collapsible PDS with annotations containing the stack the link pointed to. In addition to allowing a more straightforward definition of regularity and greatly simplifying the proofs of the paper, this model provides a more natural handling of collapse links, highlighting their connection with closures. In addition, configuration graphs of this model are isomorphic to those of collapsible PDS when restricted to configurations reachable from the initial configuration.

Our second contribution is a saturation method for backwards reachability analysis of annotated pushdown systems that can also be applied *as-is* to collapsible PDS. This is a global model-checking algorithm that is based on saturation techniques for higher-order pushdown automata [5, 15, 30]. Our algorithm handles alternating (or "two-player") as well as non-alternating systems.

In addition to the extension to annotated pushdown systems, the algorithm improves on Hague and Ong's construction for higher-order pushdown systems [15] since the number of states introduced by the construction is no longer multiplied by the number of iterations it takes to reach a fixed point, potentially leading to a large reduction in the size of the automata constructed. In addition, both the presentation and the proofs of correctness are much less involved.

Related Work In addition to the works mentioned above, solutions to global model checking problems have been proposed by Broadbent *et al.* [6]. Additionally, Salvati and Walukiewicz provide a global analysis technique for μ-calculus

[1] Kartzow and Parys have independently introduced a similar model [18].

properties using a Krivine machine model of schemes [28]. However, there are currently no versions of these algorithms available that do not pay immediately the exponential blow up.

Extensions of schemes with pattern matching have also been considered by Ong and Ramsay [25]. A recent algorithm by Kobayashi speeds up his techniques using an over-approximating least fixed point computation to give an initial input to a greatest fixed point computation [22]. Like saturation this is a 'bottom-up' approach and it would be interesting to see whether there are connections. Extensions of higher-order PDS to concurrent settings have also been considered by Seth [31].

The saturation technique has proved popular in the literature. It was introduced by Bouajjani *et al.* [4] and Finkel *et al.* [13] and based on a string rewriting algorithm by Benois [3]. It has since been extended to Büchi games [7], parity and μ-calculus conditions [16], and concurrent systems [32, 1], as well as weighted pushdown systems [27]. In addition to various implementations, efficient versions of these algorithms have also been developed [12, 33].

2 Preliminaries

2.1 Annotated Pushdown Systems

We define annotated stacks, their operations, and annotated pushdown systems.

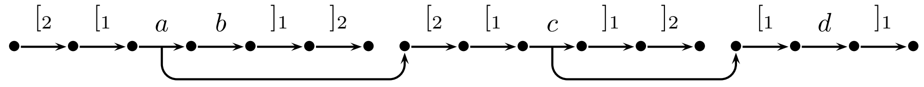
Annotated stacks Let Σ be a set of stack symbols. We define a notion of annotated higher-order stack. Intuitively, an annotated stack of order-n is an order-n stack in which stack symbols have attached annotated stacks of order at most n. For the rest of the formal definitions, we fix the maximal order to n, and use k to range between n and 1. We simultaneously define for all $1 \leq k \leq n$, the set Stacks^n_k of stacks of order-k whose symbols are annotated by stacks of order at most n. Note, we use subscripts to indicate the order of a stack.

Definition 1 (Annotated Stacks). *The family of sets* $(\mathrm{Stacks}^n_k)_{1 \leq k \leq n}$ *is the smallest family (for point-wise inclusion) such that:*

- *for all* $2 \leq k \leq n$, Stacks^n_k *is the set of all (possibly empty) sequences* $[w_1 \dots w_\ell]_k$ *with* $w_1, \dots, w_\ell \in \mathrm{Stacks}^n_{k-1}$.
- Stacks^n_1 *is all sequences* $[{a_1}^{w_1} \dots {a_\ell}^{w_\ell}]_1$ *with* $\ell \geq 0$ *and for all* $1 \leq i \leq \ell$, a_i *is a stack symbol in* Σ *and* w_i *is an annotated stack in* $\bigcup\limits_{1 \leq k \leq n} \mathrm{Stacks}^n_k$.

Observe that the above definition uses a least fixed-point. This ensures that all stacks are finite; in particular a stack cannot contain itself as an annotation. When the maximal order n is clear, we simply write Stacks_k instead of Stacks^n_k. We also write order-k stack to designate an annotated stack in Stacks^n_k.

An order-n stack can be represented naturally as an edge-labelled tree over the alphabet $\{[_{n-1}, \dots, [_1,]_1, \dots,]_{n-1}\} \uplus \Sigma$, with Σ-labelled edges having a second target to the tree representing the annotation. For technical convenience, a tree representing an order-k stack does not use $[_k$ or $]_k$ symbols (these appear uniquely at the beginning and end of the stack). An example order-3 stack is given below, with only a few annotations shown. The annotations are order-3 and order-2 respectively.



Given an order-n stack $w = [w_1 \dots w_\ell]_n$, we define $\mathit{top}_{n+1}(w) = w$ and

$$\begin{array}{rl} \mathit{top}_n([w_1 \dots w_\ell]_n) = w_1 & \text{when } \ell > 0 \\ \mathit{top}_n([]_n) = []_{n-1} & \text{otherwise} \\ \mathit{top}_k([w_1 \dots w_\ell]_n) = \mathit{top}_k(w_1) & \text{when } k < n \text{ and } \ell > 0 \end{array}$$

noting that $\mathit{top}_k(w)$ is undefined if $\mathit{top}_{k'}(w)$ is empty for any $k' > k$.

We write $u :_k v$ — where u is order-$(k-1)$ — to denote the stack obtained by placing u on top of the top_k stack of v. That is, if $v = [v_1 \dots v_\ell]_k$ then $u :_k v = [uv_1 \dots v_\ell]_k$, and if $v = [v_1 \dots v_\ell]_{k'}$ with $k' > k$, $u :_k v = [u :_k v_1, \dots, v_\ell]_{k'}$. This composition associates to the right. For example, the order-3 stack above can be written $[[[a^w b]_1]_2]_3$ and also $u :_3 v$ where u is the order-2 stack $[[a^w b]_1]_2$ and v is the empty order-3 stack $[]_3$. Then $u :_3 u :_3 v$ is $[[[a^w b]_1]_2[[a^w b]_1]_2]_3$.

Operations on Order-n Annotated Stacks The following operations can be performed on an order-n stack. We say $o \in \mathcal{O}_n$ is of order-k when k is minimal such that $o \in \mathcal{O}_k$. For example, push_k is of order k.

$$\begin{array}{rl} \mathcal{O}_n = & \{\mathit{pop}_1, \dots, \mathit{pop}_n\} \cup \{\mathit{push}_2, \dots, \mathit{push}_n\} \cup \{\mathit{collapse}_2, \dots, \mathit{collapse}_n\} \cup \\ & \left\{ \mathit{push}^1_a, \dots, \mathit{push}^n_a, \mathit{rew}_a \mid a \in \Sigma \right\} \end{array}$$

We define each stack operation for an order-n stack w. Annotations are created by push^k_a, which add a character to the top of a given stack w annotated by $\mathit{top}_{k+1}(\mathit{pop}_k(w))$. This gives a access to the context in which it was created. In Section 3.2 we give several examples of these operations.

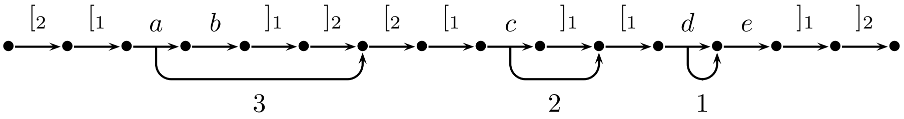
1. We set $\mathit{pop}_k(u :_k v) = v$.
2. We set $\mathit{push}_k(u :_k v) = u :_k u :_k v$.
3. We set $\mathit{collapse}_k\Big(a^{u'} :_1 u :_{(k+1)} v\Big) = u' :_{(k+1)} v$ when u is order-k and $n > k \geq 1$; and $\mathit{collapse}_n(a^u :_1 v) = u$ when u is order-n.
4. We set $\mathit{push}^k_b(w) = b^u :_1 w$ where $u = \mathit{top}_{k+1}(\mathit{pop}_k(w))$.
5. We set $\mathit{rew}_b(a^u :_1 v) = b^u :_1 v$.

Annotated Pushdown Systems We are now ready to define annotated PDS.

Definition 2 (Annotated Pushdown Systems). *An order-n* alternating annotated pushdown system (annotated PDS) *is a tuple $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R})$ where $\mathcal{P}$ is a finite set of control states, Σ is a finite stack alphabet, and $\mathcal{R} \subseteq (\mathcal{P} \times \Sigma \times \mathcal{O}_n \times \mathcal{P}) \cup \left(\mathcal{P} \times 2^{\mathcal{P}}\right)$ is a set of rules.*

We write *configurations* of an annotated PDS as a pair $\langle p, w\rangle$ where $p \in \mathcal{P}$ and $w \in \mathit{Stacks}_n$. We write $\langle p, w\rangle \longrightarrow \langle p', w'\rangle$ to denote a transition from a rule (p, a, o, p') with $\mathit{top}_1(w) = a$ and $w' = o(w)$. Furthermore, we have a transition $\langle p, w\rangle \longrightarrow \{\ \langle p', w\rangle \mid p' \in P\ \}$ whenever we have a rule $p \rightarrow P$. A non-alternating annotated PDS has no rules of this second form. We write C to denote a set of configurations.

Collapsible Pushdown Systems Annotated pushdown systems are based on collapsible PDS. In this model, stacks do not contain order-k annotations, rather they have order-k links to an order-k stack occurring lower down in the top-most order-$(k+1)$ stack. We define the model formally in the appendix. We give an example below, where links are marked with their order.



The set $\mathcal{O}_n$ is the same as in the annotated version. Collapse links are created by the push^k_a operation, which augments a with a link to pop_k of the stack being pushed onto. A $\mathit{collapse}_k$ returns to the stack that is the target of the link.

Collapsible vs. Annotated To an order-n stack w with links, we associate a canonical annotated stack $[\![\, w \,]\!]$ where each link is replaced by the annotated version of the link's target. We inductively and simultaneously define $[\![\, w \,]\!]_k$ which is the annotated stack representing $\mathit{top}_k(w)$.

$$\begin{cases} [\![\, [\,]_k :_{k+1} v \,]\!]_k = [\,]_k & \\ [\![\, u :_{k'+1} v \,]\!]_k \ \ = [\![\, u \,]\!]_{k'} :_{k'+1} [\![\, v \,]\!]_k & \text{where } 0 < k' < k \\ [\![\, a^* :_1 v \,]\!]_k \quad\ = a^{[\![\, \mathit{collapse}_{k'}(a^* :_1 v) \,]\!]_{k'}} :_1 [\![\, v \,]\!]_k & \text{where } a^* \text{ is } a \text{ with an order-}k' \text{ link} \end{cases}$$

For example, the order-3 stack above becomes $[[[a^{w_1}b]_1]_2[[c^{w_2}]_1[d^{w_3}e]_1]_2]_3$ where $w_1 = [[[c^{w_2}]_1[d^{w_3}e]_1]_2]_3$, $w_2 = [[d^{w_3}e]_1]_2$ and $w_3 = [e]_1$.

Note that some annotated stacks such as $[[a^w]_1]_2$ with $w = [[b^{[]_1}]_1]_2$ do not correspond to any stacks with links. However for all order-n stacks with links w and for any operation o of order at most n, we have $[\![\, o(w) \,]\!] = o([\![\, w \,]\!])$.

Remark 1. The configuration graphs of annotated pushdown systems of order-n are isomorphic to their collapsible counter-part when restricted to configurations reachable from the initial configuration. This implies annotated pushdown automata generate the same trees as higher-order recursion schemes, as in [6].

2.2 Regularity of Annotated Stacks

We will present an algorithm that operates on sets of configurations. For this we use order-n stack automata, thus defining a notion of regular sets of stacks. These have a nested structure based on a similar automata model by Bouajjani and Meyer [5]. The handling of annotations is similar to automata introduced by Broadbent *et al.* [6], except we read stacks top-down rather than bottom-up.

Definition 3 (Order-n Stack Automata). *An* order-n stack automaton

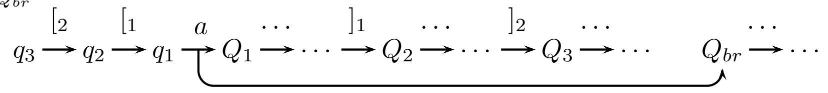
$$A = (\mathcal{Q}_n, \ldots, \mathcal{Q}_1, \Sigma, \Delta_n, \ldots, \Delta_1, \mathcal{F}_n, \ldots, \mathcal{F}_1)$$

is a tuple where Σ is a finite stack alphabet, and

1. *for all $n \geq k \geq 2$, we have $\mathcal{Q}_k$ is a finite set of states, $\Delta_k \subseteq \mathcal{Q}_k \times \mathcal{Q}_{k-1} \times 2^{\mathcal{Q}_k}$ is a transition relation, and $\mathcal{F}_k \subseteq \mathcal{Q}_k$ is a set of accepting states, and*
2. *$\mathcal{Q}_1$ is a finite set of states, $\Delta_1 \subseteq \bigcup\limits_{2 \leq k \leq n} \left(\mathcal{Q}_1 \times \Sigma \times 2^{\mathcal{Q}_k} \times 2^{\mathcal{Q}_1}\right)$ a transition relation, and $\mathcal{F}_1 \subseteq \mathcal{Q}_1$ a set of accepting states.*

Stack automata are alternating automata that read the stack in a nested fashion. Order-k stacks are recognised from states in $\mathcal{Q}_k$. A transition $(q, q', Q) \in \Delta_k$ from q to Q for some $k > 1$ can be fired when the top_{k-1} stack is accepted from $q' \in \mathcal{Q}_{(k-1)}$. The remainder of the stack must be accepted from all states in Q. At order-1, a transition (q, a, Q_{br}, Q) is a standard alternating a-transition with the additional requirement that the stack annotating a is accepted from all states in Q_{br}. A stack is accepted if a subset of $\mathcal{F}_k$ is reached at the end of each order-k stack. In the appendix, we formally define the runs of a stack automaton. We write $w \in \mathcal{L}_q(A)$ whenever w is accepted from a state q.

A (partial) run is pictured below, using $q_3 \xrightarrow{q_2} Q_3 \in \Delta_3, q_2 \xrightarrow{q_1} Q_2 \in \Delta_2$ and $q_1 \xrightarrow[Q_{br}]{a} Q_1 \in \Delta_1$. The node labelled Q_{br} begins a run on the stack annotating a.



Remark 2. In the appendix, we show several results on stack automata: membership testing is linear time; emptiness is PSPACE-complete; the sets of stacks accepted by these automata form an effective Boolean algebra (note that complementation causes a blow-up in the size of the automaton); and they accept the same family of collapsible stacks as the automata used by Broadbent *et al.* [6].

3 Algorithm

Given an annotated PDS $\mathcal{C}$ and a stack automaton A_0 with a state $q_p \in \mathcal{Q}_n$ for each control state p in $\mathcal{C}$, we define $Pre^*_{\mathcal{C}}(A_0)$ as the smallest set such that $Pre^*_{\mathcal{C}}(A_0) \supseteq \left\{ \langle p, w \rangle \;\middle|\; w \in \mathcal{L}_{q_p}(A_0) \right\}$, and

$$Pre^*_{\mathcal{C}}(A_0) \supseteq \left\{ \langle p, w \rangle \;\middle|\; \begin{array}{l} \exists \langle p, w \rangle \longrightarrow \langle p', w' \rangle \text{ with } \langle p', w' \rangle \in Pre^*_{\mathcal{C}}(A_0) \vee \\ \exists \langle p, w \rangle \longrightarrow C \text{ and } C \subseteq Pre^*_{\mathcal{C}}(A_0) \end{array} \right\}$$

recalling that C denotes a set of configurations. We build a stack automaton recognising $Pre^*_{\mathcal{C}}(A_0)$. We begin with A_0 and iterate a saturation function denoted Γ — which adds new transitions to A_0 — until a 'fixed point' has been reached. That is, we iterate $A_{i+1} = \Gamma(A_i)$ until $A_{i+1} = A_i$. As the number of states is bounded, we eventually obtain this, giving us the following theorem.

Theorem 1. *Given an alternating annotated pushdown system $\mathcal{C}$ and a stack automaton A_0, we can construct an automaton A accepting $Pre^*_{\mathcal{C}}(A_0)$.*

The construction runs in n-EXPTIME for both alternating annotated PDS and collapsible PDS — which is optimal — and can be improved to $(n-1)$-EXPTIME for non-alternating collapsible PDS when the initial automaton satisfies a certain notion of *non-alternation*, again optimal. Correctness and complexity are discussed in subsequent sections.

3.1 Notation and Conventions

Number of Transitions We assume for all $q \in \mathcal{Q}_k$ and $Q \subseteq \mathcal{Q}_k$ that there is at most one transition of the form $q \xrightarrow{q'} Q \in \Delta_k$. This condition can easily be ensured on A_0 by replacing pairs of transitions $q \xrightarrow{q_1} Q$ and $q \xrightarrow{q_2} Q$ with a single transition $q \xrightarrow{q'} Q$, where q' accepts the union of the languages of stacks accepted from q_1 and q_2. The construction maintains this condition.

Short-form Notation We introduce some short-form notation for runs. Consider the example run in Section 2.2. In this case, we write $q_3 \xrightarrow[Q_{br}]{a} (Q_1, Q_2, Q_3)$, $q_3 \xrightarrow{q_1} (Q_2, Q_3)$, and $q_3 \xrightarrow{q_2} (Q_3)$. In general, we write

$$q \xrightarrow[Q_{br}]{a} (Q_1, \ldots, Q_k) \text{ and } q \xrightarrow{q'} (Q_{k'+1}, \ldots, Q_k) .$$

In the first case, $q \in \mathcal{Q}_k$ and there exist $q_{k-1}, \ldots, q_1$ such that $q \xrightarrow{q_{k-1}} Q_k \in \Delta_k, q_{k-1} \xrightarrow{q_{k-2}} Q_{k-1} \in \Delta_{k-1}, \ldots, q_1 \xrightarrow[Q_{br}]{a} Q_1 \in \Delta_1$. Thus, we capture nested sequences of initial transitions from q. Since we assume at most one transition between any state and set of states, the intermediate states $q_{k-1}, \ldots, q_1$ are uniquely determined by q, a, Q_{br} and $Q_1, \ldots, Q_k$.

In the second case $q \in \mathcal{Q}_k$, $q' \in \mathcal{Q}_{k'}$, and there exist $q_{k-1}, \ldots, q_{k'+1}$ with $q \xrightarrow{q_{k-1}} Q_k \in \Delta_k, q_{k-1} \xrightarrow{q_{k-2}} Q_{k-1} \in \Delta_{k-1}, \ldots, q_{k'+2} \xrightarrow{q_{k'+1}} Q_{k'+2} \in \Delta_{k'+2}$ and $q_{k'+1} \xrightarrow{q'} Q_{k'+1} \in \Delta_{k'+1}$.

We lift the short-form transition notation to transitions from sets of states. We assume that state-sets $\mathcal{Q}_n, \ldots, \mathcal{Q}_1$ are disjoint. Suppose $Q = \{q_1, \ldots, q_\ell\}$ and for all $1 \leq i \leq \ell$ we have $q_i \xrightarrow[Q^i_{br}]{a} \left(Q^i_1, \ldots, Q^i_k\right)$. Then we have $Q \xrightarrow[Q_{br}]{a} (Q_1, \ldots, Q_k)$ where $Q_{br} = \bigcup_{1 \leq i \leq \ell} Q^i_{br}$ and for all k, $Q_k = \bigcup_{1 \leq i \leq \ell} Q^i_k$. Because an annotation can only be of one order, we insist that $Q_{br} \subseteq \mathcal{Q}_k$ for some k.

Finally, we remark that a transition to the empty set is distinct from having no transition.

Initial States We say a state is *initial* if it is of the form $q_p \in Q_n$ for some control state p or if it is a state $q_k \in Q_k$ for $k < n$ such that there exists a transition $q_{k+1} \xrightarrow{q_k} Q_{k+1}$ in Δ_{k+1}. We make the assumption that all initial states do not have any incoming transitions and that they are not final[2]

Adding Transitions Finally, when we add a transition $q \xrightarrow[Q_{br}]{a} (Q_1, \ldots, Q_n)$ to the automaton, then for each $n \geq k > 1$, we add $q_k \xrightarrow{q_{k-1}} Q_k$ to Δ_k (if a transition between q_k and Q_k does not already exist, otherwise we use the existing transition and state q_{k-1}) and add $q_1 \xrightarrow[Q_{br}]{a} Q_1$ to Δ_1.

3.2 The Saturation Function

Given an annotated PDS $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R})$, we define the saturation function. Examples can be found below.

Definition 4 (The Saturation Function Γ). *Given an order-n stack automaton A we define $A' = \Gamma(A)$ such that A' is A plus, for each $(p, a, o, p') \in \mathcal{R}$,*

1. *when $o = pop_k$, for each $q_{p'} \xrightarrow{q_k} (Q_{k+1}, \ldots, Q_n)$ in A, add to A'*
$$q_p \xrightarrow[\emptyset]{a} (\emptyset, \ldots, \emptyset, \{q_k\}, Q_{k+1}, \ldots, Q_n) \ ,$$
2. *when $o = push_k$, for all transitions $q_{p'} \xrightarrow[Q_{br}]{a} (Q_1, \ldots, Q_k, \ldots, Q_n)$ and $Q_k \xrightarrow[Q'_{br}]{a} (Q'_1, \ldots, Q'_k)$ in A, add to A' the transition*
$$q_p \xrightarrow[Q_{br} \cup Q'_{br}]{a} \left(Q_1 \cup Q'_1, \ldots, Q_{k-1} \cup Q'_{k-1}, Q'_k, Q_{k+1}, \ldots, Q_n\right) \ ,$$
3. *when $o = collapse_k$, when $k = n$, add $q_p \xrightarrow[\{q_{p'}\}]{a} (\emptyset, \ldots, \emptyset)$, and when $k < n$, for each transition $q_{p'} \xrightarrow{q_k} (Q_{k+1}, \ldots, Q_n)$ in A, add to A' the transition*
$$q_p \xrightarrow[\{q_k\}]{a} (\emptyset, \ldots, \emptyset, Q_{k+1}, \ldots, Q_n) \ ,$$
4. *when $o = push^k_b$ for all transitions $q_{p'} \xrightarrow[Q_{br}]{b} (Q_1, \ldots, Q_n)$ and $Q_1 \xrightarrow[Q'_{br}]{a} Q'_1$ in A with $Q_{br} \subseteq \mathcal{Q}_k$, add to A' the transition*
$$q_p \xrightarrow[Q'_{br}]{a} (Q'_1, Q_2, \ldots, Q_k \cup Q_{br}, \ldots, Q_n) \ ,$$
5. *when $o = rew_b$ for each transition $q_{p'} \xrightarrow[Q_{br}]{b} (Q_1, \ldots, Q_n)$ in A, add to A' the transition $q_p \xrightarrow[Q_{br}]{a} (Q_1, \ldots, Q_n)$.*

Finally, for every rule $p \rightarrow P$, let $Q = \{\ q_{p'} \ \mid p' \in P\ \}$, then, for each $Q \xrightarrow[Q_{br}]{a} (Q_1, \ldots, Q_n)$, add a transition $q_p \xrightarrow[Q_{br}]{a} (Q_1, \ldots, Q_n)$. For convenience, the state-sets of A' are defined implicitly from the states used in the transition relations.

[2] Hence automata cannot accept empty stacks from initial states. This can be overcome by introducing a bottom-of-stack symbol.

Examples All examples except one use the order-2 stack w', labelled by a run of a stack automaton, pictured below, where the sub-script indicates states in $\mathcal{Q}_1$ or $\mathcal{Q}_2$. Recall that the first transition of the run can be written $q_{p'} \xrightarrow[Q_{br}^2]{a} \big(Q_1^1, Q_2^1\big)$.

$$\begin{tikzcd}
q_{p'} \arrow[r, "{[_1}"] & q_1 \arrow[r, "a"] \arrow[rrrrrrrr, bend right] & Q_1^1 \arrow[r, "{]_1}"] & Q_2^1 \arrow[r, "{[_1}"] & Q_1^3 \arrow[r, "a"] \arrow[rrr, bend right] & Q_1^4 \arrow[r, "{]_1}"] & Q_2^2 & Q_{br}^1 \arrow[r, "{[_1}"] & \cdots & Q_{br}^2 \arrow[r, "{[_1}"] & \cdots
\end{tikzcd}$$

Example of (p, a, pop_2, p') Consider the stack w pictured below with $pop_2(w) = w'$. By the construction, we add a transition $q_p \xrightarrow[\emptyset]{a} (\emptyset, \{q_{p'}\})$ giving the run below labelling w, where q_1' is the state labelling the new transition $q_{p'} \xrightarrow{q_1'} \{q_{p'}\}$.

$$\begin{tikzcd}
q_p \arrow[r, "{[_1}"] & q_1' \arrow[r, "a"] & \emptyset \arrow[r, "c"] & \emptyset \arrow[r, "{]_1}"] & q_{p'} \arrow[r, "{[_1}"] & q_1 \arrow[r, "a"] \arrow[rrrr, bend right] & Q_1^1 \arrow[r, "{]_1}"] & Q_2^1 \arrow[r, "{[_1}"] & \cdots & Q_{br}^1 \arrow[r, "{[_1}"] & \cdots
\end{tikzcd}$$

Example of $(p, a, push_2, p')$ Consider the stack w below with $push_2(w) = w'$. Take $Q_2^1 \xrightarrow[Q_{br}^1]{a} \big(Q_1^4, Q_2^2\big)$ from the node labelled Q_2^1 in the run over w'. By the construction, we add $q_p \xrightarrow[Q_{br}^1 \cup Q_{br}^2]{a} \big(Q_1^1 \cup Q_1^4, Q_2^2\big)$ and obtain a run over w

$$\begin{tikzcd}
q_p \arrow[r, "{[_1}"] & q_1' \arrow[r, "a"] \arrow[rrr, bend right] & Q_1^1 \cup Q_1^4 \arrow[r, "{]_1}"] & Q_2^2 & Q_{br}^1 \cup Q_{br}^2 \arrow[r, "{[_1}"] & \cdots
\end{tikzcd} \; .$$

where q_1' is the state used by the new transition. This run combines the runs over the top two order-1 stacks of w', ensuring any stack accepted could appear twice on top of a stack already accepted. That is, $push_2(w) = w'$ is in $Pre^*_{\mathcal{C}}(A_0)$.

Example of $(p, a, collapse_2, p')$ Consider the stack w below with $collapse_2(w) = w'$. By the construction, we add a transition $q_p \xrightarrow[\{q_{p'}\}]{a} (\emptyset, \emptyset)$; hence, we have the run below, where q_1' is the state labelling the new transition $q_{p'} \xrightarrow{q_1'} \{\emptyset\}$.

$$\begin{tikzcd}
q_p \arrow[r, "{[_1}"] & q_1' \arrow[r, "a"] \arrow[rrrrrr, bend right] & \emptyset \arrow[r, "{]_1}"] & \emptyset \arrow[r, "{[_1}"] & \emptyset \arrow[r, "c"] & \emptyset \arrow[r, "{]_1}"] & \emptyset & q_{p'} \arrow[r, "{[_1}"] & q_1 \arrow[r, "a"] \arrow[rrrr, bend right] & Q_1^1 \arrow[r, "{]_1}"] & Q_2^1 \arrow[r, "{[_1}"] & \cdots & Q_{br}^1 \arrow[r, "{[_1}"] & \cdots
\end{tikzcd}$$

Example of $\big(p, a, push_b^2, p'\big)$ The stack of our running example is not constructible via a $push_b^2$ operation. Hence, we use the following stack and run for w'

$$\begin{tikzcd}
q_{p'} \arrow[r, "{[_1}"] & q_1 \arrow[r, "b"] \arrow[rrrrrrr, bend right] & Q_1^1 \arrow[r, "a"] & Q_1^2 \arrow[r, "{]_1}"] & Q_2^1 \arrow[r, "{[_1}"] & Q_1^3 \arrow[r, "a"] & Q_1^4 \arrow[r, "{]_1}"] & Q_2^2 & Q_{br} \arrow[r, "{[_1}"] & Q_1^5 \arrow[r, "a"] & Q_1^6 \arrow[r, "{]_1}"] & Q_2^3
\end{tikzcd}$$

with $q_{p'} \xrightarrow[Q_{br}]{b} \big(Q_1^1, Q_2^1\big)$ and $Q_1^1 \xrightarrow[\emptyset]{a} Q_1^2$. The construction adds $q_p \xrightarrow[\emptyset]{a} \big(Q_1^2, Q_2^1 \cup Q_{br}\big)$. This gives us a run on the stack w such that $push_b^2(w) = w'$, where q_1' is the order-1 state labelling the new order-2 transition.

$$\begin{tikzcd}
q_{p'} \arrow[r, "{[_1}"] & q_1' \arrow[r, "a"] & Q_1^2 \arrow[r, "{]_1}"] & Q_2^1 \cup Q_{br} \arrow[r, "{[_1}"] & Q_1^3 \cup Q_1^5 \arrow[r, "a"] & Q_1^4 \cup Q_1^6 \arrow[r, "{]_1}"] & Q_2^2 \cup Q_2^3
\end{tikzcd}$$

4 Correctness and Complexity

Theorem 2. *For a given $\mathcal{C}$ and A_0, let $A = A_i$ where i is the least index such that $A_{i+1} = \Gamma(A_i)$. We have $w \in \mathcal{L}_{q_p}(A)$ iff $\langle p, w\rangle \in Pre^*_{\mathcal{C}}(A_0)$.*

The proof is in the appendix. Completeness is by a straightforward induction over the "distance" to A_0. Soundness is the key technical challenge. The idea is to assign a "meaning" to each state of the automaton. For this, we define what it means for an order-k stack w to satisfy a state $q \in \mathcal{Q}_k$, which is denoted $w \models q$.

Definition 5 **($w \models q$).** *For any $Q \subseteq \mathcal{Q}_k$ and any order-k stack w, we write $w \models Q$ if $w \models q$ for all $q \in Q$, and we define $w \models q$ by a case distinction on q.*

1. *q is an initial state in $\mathcal{Q}_n$. Then for any order-n stack w, we say that $w \models q$ if $\langle q, w\rangle \in Pre^*_{\mathcal{C}}(A_0)$.*
2. *q is an initial state in $\mathcal{Q}_k$, labeling a transition $q_{k+1} \xrightarrow{q} Q_{k+1} \in \Delta_{k+1}$. Then for any order-$k$ stack w, we say that $w \models q$ if for all order-$(k+1)$ stacks v s.t. $v \models Q_{k+1}$, then $w :_{(k+1)} v \models q_{k+1}$.*
3. *q is a non-initial state in $\mathcal{Q}_k$. Then for any order-k stack w, we say that $w \models q$ if A_0 accepts w from q.*

We show the automaton constructed is sound with respect to this meaning. That is, for all $q_k \xrightarrow[Q_{br}]{a} (Q_1, \ldots, Q_k)$, we can place a^u, for any $u \models Q_{br}$, on top of any stack satisfying $Q_1, \ldots, Q_k$ and obtain a stack that satisfies q_k. By induction over the length of the stack, this property extends to complete stacks. That is, a stack is accepted from a state only if it is in its meaning. Since states q_p are assigned their meaning in $Pre^*_{\mathcal{C}}(A_0)$, we obtain soundness of the construction.

The construction is also sound for collapsible stacks. That is, $\langle p, w\rangle$ belongs to $Pre^*_{\mathcal{C}}(A_0)$ where $\mathcal{C}$ is a collapsible PDS and A_0 accepts collapsible stacks iff $\langle p, [\![w]\!]_n\rangle$ belongs to $Pre^*_{\mathcal{C}}(A_0)$ where $\mathcal{C}$ and A_0 are interpreted over annotated stacks. This is due to the commutativity of $[\![o(w)]\!] = o([\![w]\!])$.

Proposition 1. *The saturation construction for an alternating order-n annotated PDS $\mathcal{C}$ and an order-n stack automaton A_0 runs in n-EXPTIME, which is optimal.*

Proof. Let $2 \uparrow_0 \ell = \ell$ and $2 \uparrow_{i+1} \ell = 2^{2 \uparrow_i \ell}$. The number of states of A is bounded by $2 \uparrow_{(n-1)} \ell$ where ℓ is the size of $\mathcal{C}$ and A_0: each state in $\mathcal{Q}_k$ was either in A_0 or comes from a transition in Δ_{k+1}. Since the automata are alternating, there is an exponential blow up at each order except at order-n. Each iteration of the algorithm adds at least one new transition. Only $2 \uparrow_n \ell$ transitions can be added. Since the reachability problem for alternating higher-order pushdown systems is complete for n-EXPTIME [15], our algorithm is optimal.

It is known that the complexity of reachability for non-alternating collapsible PDS is in $(n-1)$-EXPTIME. The cause of the additional exponential blow up is in the alternation of the stack automata. In the appendix we show that, for a suitable notion of *non-alternating* stack automata, our algorithm can be adapted to run in $(n-1)$-EXPTIME, when the collapsible PDS is also non-alternating.

Furthermore, the algorithm is PTIME for a fixed order and number of control states. If we obtained $\mathcal{C}$ from a scheme, the number of control states is given by the arity of the scheme [14]. Since the arity and order are expected to be small, we are hopeful that our algorithm will perform well in practice.

5 Perspectives

There are several avenues of future work. First, we intend to generalise our saturation technique to computing winning regions of *parity* conditions, based on the order-1 case [17]. This will permit verification of more general specifications. We also plan to design a prototype tool to test the algorithm in practice

An important direction is that of counter example generation. When checking safety property, it is desirable to provide a trace witnessing a violation of the property. This can be used to repair the bug and as part of a *counter-example guided abstraction refinement (CEGAR)* loop enabling efficient verification algorithms. However, finding *shortest* counter examples — due to its tight connection with pumping lemmas — will present a challenging and interesting problem.

Saturation techniques have been extended to concurrent order-1 pushdown systems [32, 1]; concurrency at higher-orders would be interesting.

It will also be interesting to study notions of regularity of annotated stacks. In our notion of regularity, the forwards reachability set is not regular, due to the copy operation $push_k$. This problem was addressed by Carayol for higher-order stacks [8]; adapting these techniques to annotated PDS is a challenging problem.

References

1. M. F. Atig. Global model checking of ordered multi-pushdown systems. In *FSTTCS*, pages 216–227, 2010.
2. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
3. M. Benois *Parties rationnelles du groupe libre. Comptes-Rendus de l'Acamdémie des Sciences de Paris*, Série A:1188–1190, 1969.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, pages 135–150, 1997.
5. A. Bouajjani and A. Meyer. Symbolic Reachability Analysis of Higher-Order Context-Free Processes. In *FSTTCS*, pages 135–147 ,2004.
6. C. H. Broadbent, A. Carayol, C.-H. Luke Ong, and O. Serre. Recursion schemes and logical reflection. In *LiCS*, pages 120–129, 2010.
7. T. Cachat. *Games on Pushdown Graphs and Extensions.* PhD thesis, RWTH Aachen, 2003.
8. A. Carayol. Regular sets of higher-order pushdown stacks. In *MFCS*, pages 168–179, 2005.
9. A. Carayol and S. Wöhrle. The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *FSTTCS*, pages 112–123, 2003.
10. Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

11. W. Damm. The IO- and OI-hierarchies. *Theor. Comput. Sci.*, 20:95–207, 1982.
12. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247, 2000.
13. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Electr. Notes Theor. Comput. Sci.* 9: 27-37, 1997.
14. M. Hague, A. S. Murawski, C.-H. Luke Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LiCS*, pages 452–461, 2008.
15. M. Hague and C.-H. L. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. *Logical Methods in Computer Science*, 4(4), 2008.
16. M. Hague and C.-H. L. Ong. Analysing mu-calculus properties of pushdown systems. In *SPIN*, pages 187–192, 2010.
17. M. Hague and C.-H. L. Ong. A saturation method for the modal μ-calculus over pushdown systems. *Inf. Comput.*, 209(5):799–821, 2010.
18. A. Kartzow and P. Parys Strictness of the Collapsible Pushdown Hierarchy. arXiv:1201.3250v1 [cs.FL], 2012.
19. T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS*, pages 205–222, 2002.
20. T. Knapik, D. Niwinski, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars and panic automata. In *ICALP*, pages 1450–1461, 2005.
21. N. Kobayashi. Higher-order model checking: From theory to practice. In *LiCS*, pages 219–224, 2011.
22. N. Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *FoSSaCS*, pages 260–274, 2011.
23. A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 15:1170–1174, 1976.
24. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LiCS*, pages 81–90, 2006.
25. C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL*, pages 587–598, 2011.
26. P. Parys. Collapse operation increases expressive power of deterministic higher order pushdown automata. In *STACS*, pages 603–614, 2011.
27. T. W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.
28. S. Salvati and I. Walukiewicz. Krivine machines and higher-order schemes. In *ICALP (2)*, pages 162–173, 2011.
29. S. Schwoon. *Model-checking Pushdown Systems.* PhD thesis, Technical University of Munich, 2002.
30. A. Seth. An alternative construction in symbolic reachability analysis of second order pushdown systems. In *RP* pages 80–95, 2007.
31. A. Seth. Games on higher order multi-stack pushdown systems. In *RP*, pages 203–216, 2009.
32. D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded java programs. In *SPIN*, pages 270–287, 2008.
33. D. Suwimonteerabuth, S. Schwoon, and J. Esparza. Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In *ATVA*, pages 141–153, 2006.