View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Royal Holloway - Pure

C-SHORe

A Collapsible Approach to Verifying Higher-Order Programs

Christopher Broadbent
LIAFA, Université Paris Diderot – Paris 7 & CNRS & University of Tokyo & Technische Universität München
broadben@in.tum.de

Arnaud Carayol
LIGM, Université Paris-Est & CNRS
arnaud.carayol@univ-mlv.fr

Matthew Hague
Royal Holloway University of London & LIGM, Université Paris-Est & LIAFA, Université Paris Diderot – Paris 7 & CNRS
matthew.hague@rhul.ac.uk

Olivier Serre
LIAFA, Université Paris Diderot – Paris 7 & CNRS
olivier.serre@liafa.univ-paris-diderot.fr

Abstract

Higher-order recursion schemes (HORS) have recently received much attention as a useful abstraction of higher-order functional programs with a number of new verification techniques employing HORS model-checking as their centrepiece. This paper contributes to the ongoing quest for a truly scalable model-checker for HORS by offering a different, automata theoretic perspective. We introduce the first practical model-checking algorithm that acts on a generalisation of pushdown automata equi-expressive with HORS called *collapsible pushdown systems* (CPDS). At its core is a substantial modification of a recently studied saturation algorithm for CPDS. In particular it is able to use information gathered from an approximate forward reachability analysis to guide its backward search. Moreover, we introduce an algorithm that prunes the CPDS prior to model-checking and a method for extracting counter-examples in negative instances. We compare our tool with the state-of-the-art verification tools for HORS and obtain encouraging results. In contrast to some of the main competition tackling the same problem, our algorithm is fixed-parameter tractable, and we also offer significantly improved performance over the only previously published tool of which we are aware that also enjoys this property. The tool and additional material are available from `http://cshore.cs.rhul.ac.uk`.

Categories and Subject Descriptors F.1.1 [*Models of Computation*]: Automata

Keywords Higher-Order; Verification; Model-Checking; Recursion Schemes; Collapsible Pushdown Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP '13, September 25 - 27 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500589

1. Introduction

Functional languages such as Haskell, OCaML and Scala strongly encourage the use of higher-order functions. This represents a challenge for software verification, which usually does not model recursion accurately, or models only first-order calls (e.g. SLAM [2] and Moped [33]). However, there has recently been much interest in a model called *higher-order recursion schemes (HORS)* (see e.g. [29]), which offers a way of abstracting functional programs in a manner that precisely models higher-order control-flow.

The execution trees of HORS enjoy decidable μ-calculus theories [29], which testifies to the good algorithmic properties of the model. Even 'reachability' properties (subsumed by the μ-calculus) are very useful in practice. As a simple example, the safety of incomplete pattern matching clauses could be checked by asking whether the program can 'reach a state' where a pattern match failure occurs. More complex 'reachability' properties can be expressed using a finite automaton and could, for example, specify that the program respects a certain discipline when accessing a particular resource (see [22]). Despite even reachability being $(n-1)$-EXPTIME complete, recent research has revealed that useful properties of HORS can be checked in practice.

Kobayashi's TRecS [23] tool, which checks properties expressible by a deterministic trivial Büchi automaton (all states accepting), was the first to achieve this. It works by determining whether a HORS is typable in an intersection-type system characterising the property to be checked [22]. In a bid to improve scalability, a number of other algorithms have subsequently been designed and implemented such as Kobayashi *et al.*'s GTRecS(2) [25, 26] and Neatherway *et al.*'s TravMC [28] tools (appearing in ICFP 2012), all of which remain based on intersection type inference.

This work is the basis of various techniques for verifying functional programs. Kobayashi *et al.* have developed MoCHi [27] that checks safety properties of (OCaML) programs, and EHMTT Verifier [38] for tree processing programs. Both use a model-checker for recursion schemes as a central component. Similarly, Ramsay and Ong [30] provide a verification procedure for programs with pattern matching employing recursion schemes as an abstraction.

Despite much progress, even the state-of-the-art TRecS does not scale to recursion schemes big enough to model realistically sized programs; achieving scalability while accurately tracking higher-order control-flow is a challenging problem. This paper of-

fers an automata-theoretic perspective on this challenge, providing a fresh set of tools that contrast with previous intersection-type approaches. Techniques based on pushdown automata have previously visited ICFP, such as the approximate higher-order control-flow analysis CFA2 [40], but our aims are a bit different in that we wish to match the expressivity of HORS. Consequently we require a more sophisticated notion of pushdown automaton.

Collapsible pushdown systems (CPDS) [16] are an alternative representation of the class of execution trees that can be generated by recursion schemes (with linear-time mutual-translations between the two formalisms [10, 16]). While pushdown systems augment a finite-state machine with a stack and provide an ideal model for first-order programs [20], collapsible pushdown systems model higher-order programs by extending the stack of a pushdown system to a nested "stack-of-stacks" structure. The nested stack structure enables one to represent closures. Indeed the reader might find it helpful to view a CPDS as being Krivine's Abstract Machine in a guise making it amenable to the generalisation of techniques for pushdown model-checking. Salvati and Walukiewicz have studied in detail the connection with the Krivine abstract machine [32].

For ordinary ('order-1') pushdown systems, a model-checking approach called *saturation* has been successfully implemented by tools such as Moped [33] and PDSolver [15]. Given a regular set of configurations of the pushdown system (represented by a finite automaton A acting on stacks), saturation can solve the 'backward reachability problem' by computing another finite automaton recognising a set of configurations from which a configuration in $\mathcal{L}(A)$ can be reached. This is a fixed-point computation that gradually adds transitions to A until it is 'saturated'. If A recognises a set of error configurations, one can determine whether the pushdown system is 'safe' by checking if its initial configuration is recognised by the automaton computed by saturation.

We recently extended the saturation method to a backward reachability analysis of collapsible pushdown systems [7]. This runs in PTIME when the number of control states is bounded. Crucially, this condition is satisfied when translating from recursion schemes of bounded arity with properties represented by automata of bounded size [16]. Whilst the HORS/intersection-type based tool GTRecS(2) also enjoys this fixed-parameter tractability (others do not), it times out on many benchmarks that our tool solves quickly.

Motivated by these facts, we revisit the foundations of higher-order verification tools and introduce C-SHORe — the first model-checking tool for the (direct) analysis of collapsible pushdown systems. Whilst the tool is based on the ICALP 2012 result, some substantial modifications and additions are made to the algorithm, leading to several novel practical and theoretical contributions:

1. An approximate *forward* reachability algorithm providing data
   (a) . . . allowing for the CPDS to be pruned so that saturation receives a smaller input.
   (b) . . . employed by a modified saturation algorithm to guide its *backward* search.

   This is essential for termination on most of our benchmarks.
2. A method for extracting witnesses to reachability.
3. A complete reworking of the saturation algorithm that speeds up the fixed-point computation.
4. Experimental results showing our approach compares well with TRecS, GTRecS(2) and TravMC.

It is worth remarking that the other type-based tools mentioned above all work by propagating information in a forward direction with respect to the evaluation of the model. In contrast, the raw saturation algorithm works backwards, but we also show here how forward and backward propagation can be combined.

In Sections 5 to 8 we describe the original contributions of this paper. These sections can be understood independently of one another, and hence the reader does not need to fully grasp each section before continuing to the next. The remaining sections describe the background, related work and conclusions.

In Section 3 we recall the basic structures used in the paper as well as recapping the ICALP 2012 algorithm. In Section 5 we describe the approximate forwards-reachability analysis and how it is exploited. In Section 6 we show how to generate witnesses to reachability. In Section 7, we then consider how to restructure the saturation algorithm to more efficiently compute the fixed-point. We provide experimental results in Section 8. Note that we do not discuss in formal detail the translation from HORS model-checking to reachability for CPDS, which essentially follows [10]. However, we do give an informal overview in Section 2, which we hope serves to demonstrate how closures can be accurately modelled.

The tool is available at `http://cshore.cs.rhul.ac.uk`. An appendix containing additional material appears in the long version of this paper, available from the same URL.

2. Modelling Higher-Order Programs

In this section we give an informal introduction to the process of modelling higher-order programs for verification. In particular, we show how a simple example program can be modelled using a higher-order recursion scheme, and then we show how this scheme is evaluated using a collapsible pushdown system. For a more systematic approach to modelling higher-order programs with recursion schemes, we refer the reader to work by Kobayashi *et al.* [27]. This section is for background only, and can be safely skipped.

For this section, consider the toy example below.

```
Main = MakeReport Nil
MakeReport x = if * (Commit x)
               else (AddData x MakeReport)
AddData y f = if * (f Error) else (f Cons(_, y))
```

In this example, `*` represents a non-deterministic choice (that may, for example, be a result of some input by the user). Execution begins at the `Main` function whose aim is to make a report which is a list. We begin with an empty report and send it to `MakeReport`. Either `MakeReport` indicates the report is finished and commits the report somehow, or it adds an item to the head of the list, using the `AddData` function, which takes the report so far, and a continuation. `AddData` either detects a problem with the new data (maybe it is inconsistent with the rest of the report) and flags an error by passing `Error` to the continuation, or extends the report with some item. In this case, the programmer has not provided error handling as part of the `MakeReport` function, and so an `Error` may be committed.

2.1 Higher-Order Recursion Schemes

As a first step in modelling this program, we introduce, informally, higher-order recursion schemes. These are rewrite systems that generate the computation tree of a functional program. A rewrite rule takes the form

$$N \, \phi \, x \; \hookrightarrow \; t$$

where N is a typed non-terminal with (possibly higher-order) arguments ϕ and x. A term $N \; t_\phi \; t_x$ rewrites to t with t_ϕ substituted for ϕ and t_x substituted for x. Note that recursion schemes require t to be of ground type. We will illustrate the behaviour of a recursion scheme and its use in analysis using the toy example from above.

We can directly model our example with the scheme

$$\begin{array}{rcl} main & \hookrightarrow & M \; nil \\ M \; x & \hookrightarrow & or \; (commit \; x) \; (A \; x \; M) \\ A \; y \; \phi & \hookrightarrow & or \; (\phi \; error) \; (\phi \; (cons \; y)) \end{array}$$

where M is the non-terminal associated with the `MakeReport` function, and A is the non-terminal associated with the `AddData` function; nil, or, $commit$, $error$ and $cons$ are terminal symbols of arity 0, 2, 1, 0 and 1 respectively (e.g. in the second rule, *or* takes the two arguments $(commit\ x)$ and $(A\ x\ M)$). The scheme above begins with the non-terminal $main$ and, through a sequence of rewrite steps, generates a tree representation of the evolution of the program. Figure 1, described below, shows such a sequence.

Beginning with the non-terminal $main$, we apply the first rewrite rule to obtain the tree representing the term $(A\ nil)$. We then apply the second rewrite rule, instantiating x with nil to obtain the next tree in the sequence. This continues *ad infinitum* to produce a possibly infinite tree labelled only by terminals.

We are interested in ensuring the correctness of the program. In our case, this means ensuring that the program never attempts to $commit$ an $error$. By inspecting the rightmost tree in Figure 1, we can identify a branch labelled $or, or, or, commit, error$. This is an error situation because $commit$ is being called with an $error$ report. In general we can define the regular language $\mathcal{L}_{err} = or^* commit\ or^* error$. If the tree generated by the recursion scheme contains a branch labelled by a word appearing in $\mathcal{L}_{err}$, then we have identified an error in the program.

2.2 Collapsible Pushdown Automata

Previous research into the verification of recursion schemes has used an approach based on *intersection types* (e.g. [24, 28]). In this work we investigate a radically different approach exploiting the connection between higher-order recursion schemes and an automata model called *collapsible pushdown automata* (CPDA). These two formalisms are, in fact, equivalent.

Theorem 2.1 (Equi-expressivity [16]). *For each order-n recursion scheme, there is an order-n collapsible pushdown automaton generating the same tree, and vice-versa. Furthermore, the translations in both directions are linear.*

We describe at a high level the structure of a CPDA and how they can be used to evaluate recursion schemes. In our case, this means outputting a sequence of non-terminals representing each path in the tree. More formal definitions are given in Section 3. At any moment, a CPDA is in a *configuration* $\langle p, w \rangle$, where p is a control state taken from a finite set $\mathcal{P}$, and w is a higher-order collapsible stack. In the following we will focus on the stack. Control states are only needed to ensure that sequences of stack operations occur in the correct order and are thus elided for clarity.

In the case of our toy example, we have an order-2 recursion scheme and hence an order-2 stack. An order-1 stack is a stack of characters a from a finite alphabet Σ. An order-2 stack is a stack of order-1 stacks. Thus we can write $[[main]]$ to denote the order-2 stack containing only the order-1 stack $[main]$; $[main]$ is an order-1 stack containing only the character $main$. In general Σ will contain all subterms appearing in the original statement of our toy example recursion scheme. The evolution of the CPDA stack is given in Figure 2 and explained below.

The first step is to rewrite $main$ using $main \hookrightarrow M\ nil$. Since $(M\ nil)$ is a subterm of our recursion scheme, we have $(M\ nil) \in \Sigma$ and we simply rewrite the stack $[[main]]$ to $[[M\ nil]]$.

The next step is to call the function M. As is typical in the execution of programs, a function call necessitates a new stack frame. In particular, this means pushing the body of M (that is $(or\ (commit\ x)\ (A\ x\ M)))$ onto the stack, resulting in the third stack in Figure 2. Note that we do not instantiate the variable x, hence we use only the subterms appearing in the recursion scheme.

Recall that we want to obtain a CPDA that outputs a sequence of terminals representing each path in the tree. To evaluate the term *or* $(\cdots)\ (\cdots)$ we have to output the terminal *or* and then (non-deterministically) choose a branch of the tree to follow. Let us choose $(A\ x\ M)$. Hence, the CPDA outputs the terminal *or* and rewrites the top term to $(A\ x\ M)$. Next we make a call to the A function, pushing its body on to the stack, and then pick out the $(\phi\ error)$ branch of the *or* terminal. This takes us to the beginning of the second row of Figure 2.

To proceed, we have to evaluate $(\phi\ error)$. To be able to do this, we have to know the value of ϕ. We can obtain this information by inspecting the stack and seeing that the second argument of the call of A is M. However, since we can only see the top of a stack, we would have to remove the character $(\phi\ error)$ to be able to determine that $\phi = M$, thus losing our place in the computation.

This is where we use the power of order-2 stacks. An order-2 stack is able — via a $push_2$ operation — to create a copy of its topmost order-1 stack. Hence, we perform this copy (note that the top of the stack is written on the left) and delve into the copy of the stack to ascertain the value of ϕ. While doing this we also create a *collapse link*, pictured as an arrow from M to the term $(\phi\ error)$. This collapse link is a pointer from M to the context in which M will be evaluated. In particular, if we need to know the value of x in the body of M, we will need to know that M was called with the $error$ argument, within the term $(\phi\ error)$; the collapse link provides a pointer to this information (in other words we have encoded a closure in the stack). We can access this information via a *collapse* operation. These are the two main features of a higher-order collapsible stack, described formally in the next section.

To continue the execution, we push the body of M on to the stack, output the *or* symbol and choose the $(commit\ x)$ branch. Since $commit$ is a terminal, we output it and pick out x for evaluation. To know the value of x, we have to look into the stack and follow the collapse link from M to $(\phi\ error)$. Note that we do not need to create a copy of the stack here because x is an order-0 variable and thus represents a self-contained execution. Since $error$ is the value of the argument we are considering, we pick it out and then output it before terminating. This completes the execution corresponding to the error branch identified in Figure 1.

2.3 Collapsible Pushdown Systems

The CPDA output $or, or, or, commit, error$ in the execution above. This is an error sequence in $\mathcal{L}_{err}$ and should be flagged. In general, we take the finite automaton A representing the regular language $\mathcal{L}_{err}$ and form a product with the CPDA described above. This results in a CPDA that does not output any symbols, but instead keeps in its control state the progression of A. Thus we are interested in whether the CPDA is able to reach an accepting state of A, not the language it generates. We call a CPDA without output symbols a *collapsible pushdown system* (CPDS), and the question of whether a CPDS can reach a given state is the reachability problem. This is the subject of the remainder of the paper.

3. Preliminaries

3.1 Collapsible Pushdown Systems

We first introduce higher-order collapsible stacks and their operations, before giving the definition of collapsible pushdown systems.

3.1.1 Higher-Order Collapsible Stacks and Their Operations

Higher-order collapsible stacks are built from a stack alphabet Σ and form a nested "stack-of-stacks" structure. Using an idea from *panic automata* [21], each stack character contains a pointer — called a "link" — to a position lower down in the stack. Operations updating stacks (defined below) may create copies of sub-stacks. The link is intuitively a pointer to the context in which the stack character was first created. In the sequel, we fix the maximal order to n, and use k to range between 1 and n. In the definition below, we defer the meaning of *collapse link* to Definition 3.2.

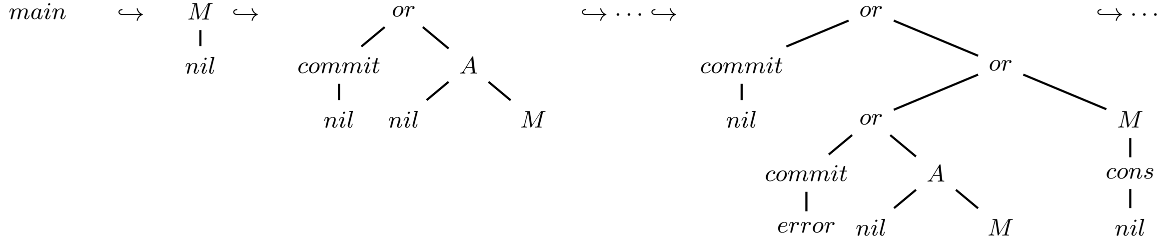

Figure 1: The behaviour of a toy recursion scheme.

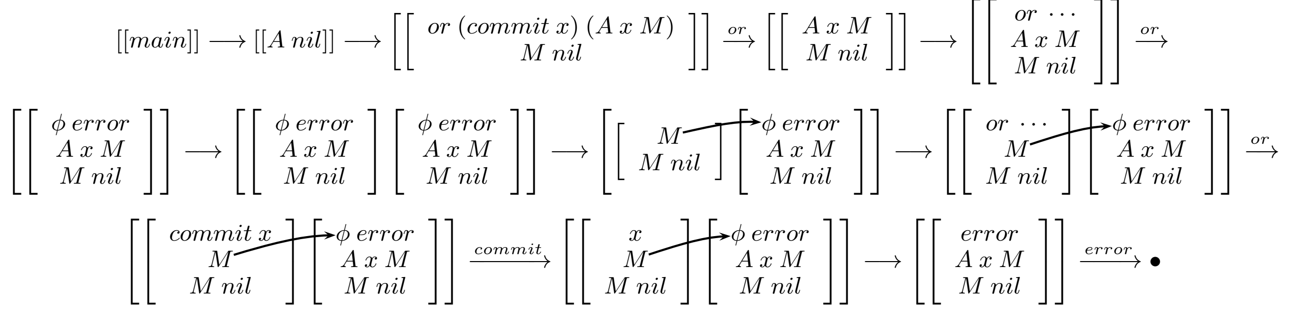

Figure 2: A stack evaluating the toy example.

Definition 3.1 (Order-n Collapsible Stacks)**.** *Given a finite set of stack characters* Σ*, an* order-0 stack *is simply a character* $a \in \Sigma$*. An* order-n stack *is a sequence* $w = [w_\ell \dots w_1]_n$ *such that each* w_i *is an order-*$(n-1)$ *stack and each character* a *on the stack is augmented with a collapse link. The top-most stack is* w_ℓ*. Let* $Stacks_n$ *denote the set of order-n stacks.*

Collapse links point to positions in the stack. Before describing them formally, we give an informal description and some basic definitions. An order-n stack can be represented naturally as an edge-labelled word-graph over the alphabet $\{[_{n-1}, \dots, [_1,]_1, \dots,]_{n-1}\} \uplus \Sigma$, with additional collapse-links pointing from a stack character in Σ to the beginning of the graph representing the target of the link. For technical convenience we do not use $[_n$ or $]_n$ symbols (these appear uniquely at the beginning and end of the stack). An example order-3 stack is given in Figure 3, with only a few collapse links shown, ranging from order-3 to order-1 respectively.

Stacks are written with the top on the left. Given an order-n stack $[w_\ell \dots w_1]_n$, we define

$$\begin{array}{rcll} top_{n+1}([w_\ell \dots w_1]_n) & = & [w_\ell \dots w_1]_n & \\ top_n([w_\ell \dots w_1]_n) & = & w_\ell & \text{if } \ell > 0 \\ top_n([\,]_n) & = & [\,]_{n-1} & \text{otherwise} \\ top_k([w_\ell \dots w_1]_n) & = & top_k(w_\ell) & \text{if } k < n \text{ and } \ell > 0 \end{array}$$

noting that $top_k(w)$ is undefined if $top_{k'}(w)$ is empty for any $k' > k$. We also remove the top portion of a top_k stack using

$$bot^i_n([w_\ell \dots w_1]_n) = [w_i \dots w_1]_n$$

when $i \leq \ell$ and $\ell > 0$, and

$$bot^i_k([w_\ell \dots w_1]_n) = [bot^i_k(w_\ell) w_{\ell-1} \dots w_1]_n$$

when $k < n$ and $\ell > 0$. We are now ready to define collapse links.

Definition 3.2 (Collapse Links)**.** *An* order-k collapse link *is a pair* (k, i) *where* $1 \leq k \leq n$ *and* $i > 0$*.*

For $top_1(w) = a$ where a has the link (k, i), the destination of the link is $bot^i_k(w)$. We disallow collapse links where bot^i_k does not lead to a valid stack. The example stack in Figure 3 is thus $[[[a^{(3,1)}b]_1]_2[[c^{(2,1)}]_1[d^{(1,1)}e]_1]_2]_3$, where collapse links are denoted as superscripts. Often (as we have done for the stack characters b and e) we will omit these superscripts for readability.

Finally the following notation appends a stack on top of another. Given an order-k stack $v = [v_\ell \dots v_1]_k$ and an order-$(k'-1)$ stack u (with $k' \leq k$), we define $u :_{k'} v = [u\, v_\ell \dots v_1]_k$ if $k' = k$ and $u :_{k'} v = [(u :_{k'} v_\ell) \dots v_1]_k$ if $k' < k$.

The following operations apply to an order-n collapsible stack.

$$\begin{array}{rcl} \mathcal{O}_n & = & \{pop_1, \dots, pop_n\} \cup \{push_2, \dots, push_n\} \cup \\ & & \{collapse_2, \dots, collapse_n\} \cup \\ & & \left\{push^2_a, \dots, push^n_a, rew_a \mid a \in \Sigma\right\} \end{array}$$

We define each stack operation for an order-n stack w. Collapse links are created by $push^k_a$, which add a character to the top of a given stack w with a link pointing to $top_{k+1}(pop_k(w))$. This gives a access to the context in which it was created. We set

1. $pop_k(u :_k v) = v$,
2. $push_k(u :_k v) = u :_k (u :_k v)$,
3. $collapse_k(w) = bot^i_k(w)$ where $top_1(w) = a^{(k,i)}$ for some i,
4. $push^k_b(w) = b^{(k,\ell-1)} :_1 w$ where $top_{k+1}(w) = [w_\ell \dots w_1]_k$,
5. $rew_b\left(a^{(k,i)} :_1 v\right) = b^{(k,i)} :_1 v$.

Note that, for a $push_k$ operation, links outside of $u = top_k(w)$ point to the same destination in both copies of u, while links pointing within u point within the respective copies of u. Since $collapse_1$ would always be equivalent to pop_1, we neither create nor follow order-1 links. (Often in examples we do not illustrate links that are never used.) For a more detailed introduction see [10].

3.1.2 Collapsible Pushdown Systems

We are now ready to define collapsible pushdown systems.

Definition 3.3 (Collapsible Pushdown Systems)**.** *An order-*n collapsible pushdown system (n-CPDS) *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* $\mathcal{R} \subseteq \mathcal{P} \times \Sigma \times \mathcal{O}_n \times \mathcal{P}$ *is a set of rules.*

A *configuration* of a CPDS is a pair $\langle p, w\rangle$ where $p \in \mathcal{P}$ and $w \in Stacks_n$. We denote by $\langle p, w\rangle \longrightarrow \langle p', w'\rangle$ a transition from a rule (p, a, o, p') with $top_1(w) = a$ and $w' = o(w)$. A run of a CPDS is a finite sequence $\langle p_0, w_0\rangle \longrightarrow \dots \longrightarrow \langle p_\ell, w_\ell\rangle$.

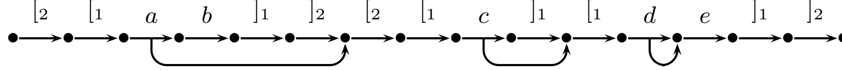

Figure 3: A graph representation of a stack.

3.2 Representing Sets of Stacks

Our algorithm represents sets of configurations using order-n stack automata. These are a kind of alternating automata with a nested structure that mimics the nesting in a higher-order collapsible stack. We recall the definition below.

Definition 3.4 (Order-n Stack Automata)**.** *An* order-n stack automaton $A = (\mathbb{Q}_n, \ldots, \mathbb{Q}_1, \Sigma, \Delta_n, \ldots, \Delta_1, \mathcal{F}_n, \ldots, \mathcal{F}_1)$ *is a tuple where* Σ *is a finite stack alphabet, and*

1. *for all* $n \geq k \geq 2$*, we have* $\mathbb{Q}_k$ *is a finite set of states,* $\mathcal{F}_k \subseteq \mathbb{Q}_k$ *is a set of accepting states, and* $\Delta_k \subseteq \mathbb{Q}_k \times \mathbb{Q}_{k-1} \times 2^{\mathbb{Q}_k}$ *is a transition relation such that for all* q *and* Q *there is* at most one q' *with* $(q, q', Q) \in \Delta_k$*, and*
2. $\mathbb{Q}_1$ *is a finite set of states,* $\mathcal{F}_1 \subseteq \mathbb{Q}_1$ *a set of accepting states, and* $\Delta_1 \subseteq \bigcup_{2 \leq k \leq n} \left(\mathbb{Q}_1 \times \Sigma \times 2^{\mathbb{Q}_k} \times 2^{\mathbb{Q}_1}\right)$ *a transition relation.*

The sets $\mathbb{Q}_k$ are disjoint and their states recognise order-k stacks. Stacks are read from "top to bottom". A transition $(q, q', Q) \in \Delta_k$, written $q \xrightarrow{q'} Q$, from q to Q for some $k > 1$ requires that the top_{k-1} stack is accepted from $q' \in \mathbb{Q}_{(k-1)}$ and the rest of the stack is accepted from each state in Q. At order-1, a transition (q, a, Q_{col}, Q) has the additional requirement that the stack linked to by a is accepted from Q_{col}. A stack is accepted if a subset of $\mathcal{F}_k$ is reached at the end of each order-k stack. We write $w \in \mathcal{L}_q(A)$ to denote the set of all w accepted from q. Note that a transition to the empty set is distinct from having no transition. Figure 4 shows part of a run over the stack in Figure 3 where each node in the graph is labelled by the states from which the remainder of the stack containing it (as well as the stacks linked to) must be accepted. Note, e.g., that since Q_2 appears at the bottom of an order-2 stack, we must have $Q_2 \subseteq \mathcal{F}_2$ for the run to be accepting. The transitions used are $q_3 \xrightarrow{q_2} Q_3 \in \Delta_3$, $q_2 \xrightarrow{q_1} Q_2 \in \Delta_2$, and $q_1 \xrightarrow[Q_{col}]{a} Q_1 \in \Delta_1$. See Section 4 for further examples.

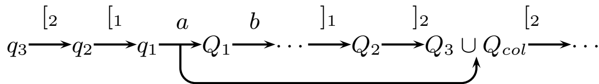

Figure 4: Part of a run over an example stack.

3.2.1 Representing Transitions and States

We use a *long-form* notation (defined below) that captures nested sequences of transitions. For example, we may write $q_3 \xrightarrow[Q_{col}]{a} (Q_1, Q_2, Q_3)$ to capture the transitions shown in Figure 4. Together, these indicate that after starting from the beginning of the stack and reading only the topmost stack character, the remainder of the stack must be accepted by Q_{col}, Q_1, Q_2, and Q_3. More generally, we may also use $q_3 \xrightarrow{q_1} (Q_2, Q_3)$, and $q_3 \xrightarrow{q_2} (Q_3)$.

Formally, when $q \in \mathbb{Q}_k$, $q' \in \mathbb{Q}_{k'}$, $Q_i \subseteq \mathbb{Q}_i$ for all $k \geq i \geq 1$, and there is some i with $Q_{col} \subseteq \mathbb{Q}_i$, we write

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

In the first case, there exist $q_{k-1}, \ldots, q_1$ such that $q \xrightarrow{q_{k-1}} Q_k \in \Delta_k, q_{k-1} \xrightarrow{q_{k-2}} Q_{k-1} \in \Delta_{k-1}, \ldots, q_1 \xrightarrow[Q_{col}]{a} Q_1 \in \Delta_1$. In the second case 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 $q_{k'+1} \xrightarrow{q'} Q_{k'+1} \in \Delta_{k'+1}$.

Remark 3.1. *We may also write* $q_{Q_k, \ldots, Q_{k'+1}}$ *for the* q' *above (which is uniquely determined by* $Q_k, \ldots, Q_{k'+1}$*).*

Note that our definitions mean that we have, e.g., $q \xrightarrow[Q_{col}]{a} (Q_1, Q_2, Q_3)$ if and only if we have $q_{Q_3, Q_2} \xrightarrow[Q_{col}]{a} Q_1$ in Δ_1.

3.2.2 Representing Sets of Transitions

Let Δ_k^S denote the set of all order-k *long-form* transitions $q \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_k)$ of order-k. For a set $T = \{t_1, \ldots, t_\ell\} \subseteq \Delta_k^S$, we say T is of the form

$$Q \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_k)$$

whenever $Q = \{q_1, \ldots, q_\ell\}$ and for all $1 \leq i \leq \ell$ we have $t_i = q_i \xrightarrow[Q^i_{col}]{a} (Q^i_1, \ldots, Q^i_k)$ and $Q_{col} = \bigcup_{1 \leq i \leq \ell} Q^i_{col}$ and for all $1 \leq k' \leq k$, $Q_{k'} = \bigcup_{1 \leq i \leq \ell} Q^i_{k'}$. Because a link can only be of one order, we insist that $Q_{col} \subseteq \mathbb{Q}_{k'}$ for some $1 \leq k' \leq n$.

3.3 Representing Sets of Configurations

We define a notion of $\mathcal{P}$-multi-automata [4] for representing sets of configurations of collapsible pushdown systems.

Definition 3.5 ($\mathcal{P}$-Multi Stack Automata)**.** *Given an order-n CPDS with control states $\mathcal{P}$, a $\mathcal{P}$-multi stack automaton is an order-n stack automaton* $A = (\mathbb{Q}_n, \ldots, \mathbb{Q}_1, \Sigma, \Delta_n, \ldots, \Delta_1, \mathcal{F}_n, \ldots, \mathcal{F}_1)$ *such that for each* $p \in \mathcal{P}$ *there exists a state* $q_p \in \mathbb{Q}_n$*.*

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}. The language of a $\mathcal{P}$-multi stack automaton A is the set $\mathcal{L}(A) = \left\{ \langle p, w \rangle \;\middle|\; w \in \mathcal{L}_{q_p}(A) \right\}$.

3.4 Basic Saturation Algorithm

Our algorithm computes the set $Pre^*_{\mathcal{C}}(A_0)$ of a collapsible pushdown system $\mathcal{C}$ and a $\mathcal{P}$-multi stack automaton A_0. We assume without loss of generality that initial states of A_0 do not have incoming transitions and are not final. To accept empty stacks from initial states, a bottom-of-stack symbol can be used.

Let $Pre^*_{\mathcal{C}}(A_0)$ be the smallest set with $Pre^*_{\mathcal{C}}(A_0) \supseteq \mathcal{L}(A_0)$, and $Pre^*_{\mathcal{C}}(A_0) \supseteq \{\langle p, w\rangle \mid \exists \langle p, w\rangle \longrightarrow \langle p', w'\rangle \in Pre^*_{\mathcal{C}}(A_0)\}$.

We begin with A_0 and iterate a saturation function Π — adding new transitions to A_0 — until a 'fixed point' is reached; that is, we cannot find any more transitions to add.

Notation for Adding Transitions During saturation we designate transitions $q_n \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$ to be added to the automaton. Recall this represents $q \xrightarrow{q_{n-1}} Q_n \in \Delta_n, q_{n-1} \xrightarrow{q_{n-2}} Q_{n-1} \in \Delta_{n-1}, \ldots, q_1 \xrightarrow[Q_{col}]{a} Q_1 \in \Delta_1$. Hence, we first, for each $n \geq k > 1$, add $q_k \xrightarrow{q_{k-1}} Q_k$ to Δ_k if it does not already exist. Then, we add $q_1 \xrightarrow[Q_{col}]{a} Q_1$ to Δ_1.

Justified Transitions In this paper, we extend the saturation function to add *justifications* to new transitions that indicate the provenance of each new transition. This permits counter example gen-

eration. To each $t = q \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$ we will define the justification $J(t)$ to be either 0 (indicating the transition is in A_0), a pair (r, i), a tuple (r, t', i) or a tuple (r, t', T, i) where r is a rule of the CPDS, i is the number of iterations of the saturation function required to introduce the transition, t' is a long-form transition and T is a set of such transitions. This information will be used in Section 6 for generating counter examples. Note that we apply J to the long-form notation. In reality, we associate each justification with the unique order-1 transition $q_1 \xrightarrow[Q_{col}]{a} Q_1$ associated to each t.

The Saturation Function We are now ready to recall the saturation function Π for a given $\mathcal{C} = (\mathcal{P}, \Sigma, \mathcal{R})$. As described above, we apply this function to A_0 until a fixed point is reached. First set $J(t) = 0$ for all transitions of A_0. The intuition behind the saturation rules can be quickly understood via a rewrite rule (p, a, rew_b, p') which leads to the addition of a transition $q_p \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$ whenever there already existed a transition $q_{p'} \xrightarrow[Q_{col}]{b} (Q_1, \ldots, Q_n)$. Because the rewrite can change the control state from p to p' and the top character from a to b, we must have an accepting run from q_p with a on top whenever we had an accepting run from $q_{p'}$ with b on top. We give examples and intuition of the more complex steps in Section 4, which may be read alongside the definition below.

Definition 3.6 (The Saturation Function Π)**.** *Given an order-n stack automaton A_i we define $A_{i+1} = \Pi(A_i)$. The state-sets of A_{i+1} are defined implicitly by the transitions which are those in A_i plus, for each $r = (p, a, o, p') \in \mathcal{R}$,*

1. *when $o = pop_k$, for each $q_{p'} \xrightarrow{q_k} (Q_{k+1}, \ldots, Q_n)$ in A_i, add $t = q_p \xrightarrow[\emptyset]{a} (\emptyset, \ldots, \emptyset, \{q_k\}, Q_{k+1}, \ldots, Q_n)$ to A_{i+1} and set $J(t) = (r, i+1)$ whenever t is not already in A_{i+1},*
2. *when $o = push_k$, for each $t = q_{p'} \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_k, \ldots, Q_n)$ and T of the form $Q_k \xrightarrow[Q'_{col}]{a} (Q'_1, \ldots, Q'_k)$ in A_i, add to A_{i+1} the transition*
$$t' = q_p \xrightarrow[Q_{col} \cup Q'_{col}]{a} \begin{pmatrix} Q_1 \cup Q'_1, \ldots, Q_{k-1} \cup Q'_{k-1}, \\ Q'_k, \\ Q_{k+1}, \ldots, Q_n \end{pmatrix}$$
and set $J(t') = (r, t, T, i+1)$ if t' is not already in A_{i+1},
3. *when $o = collapse_k$, when $k = n$, add $t = q_p \xrightarrow[\{q_{p'}\}]{a} (\emptyset, \ldots, \emptyset)$ if it does not exist, and when $k < n$, for each transition $q_{p'} \xrightarrow{q_k} (Q_{k+1}, \ldots, Q_n)$ in A_i, add to A_{i+1} the transition $t = q_p \xrightarrow[\{q_k\}]{a} (\emptyset, \ldots, \emptyset, Q_{k+1}, \ldots, Q_n)$ if t does not already exist. In all cases, if t is added, set $J(t) = (r, i+1)$,*
4. *when $o = push^k_b$ for all transitions $t = q_{p'} \xrightarrow[Q_{col}]{b} (Q_1, \ldots, Q_n)$ and $T = Q_1 \xrightarrow[Q'_{col}]{a} Q'_1$ in A_i with $Q_{col} \subseteq \mathbb{Q}_k$, add to A_{i+1} the transition*
$$t' = q_p \xrightarrow[Q'_{col}]{a} \left(Q'_1, Q_2, \ldots, Q_k \cup Q_{col}, \ldots, Q_n\right) ,$$
and set $J(t') = (r, t, T, i+1)$ if t' is not already in A_{i+1},
5. *when $o = rew_b$ for each transition $t = q_{p'} \xrightarrow[Q_{col}]{b} (Q_1, \ldots, Q_n)$ in A_i, add to A_{i+1} the transition $t' = q_p \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_n)$, setting $J(t') = (r, t, i)$ when t' is not already in A_{i+1}.*

From A_0, we iterate $A_{i+1} = \Pi(A_i)$ until $A_{i+1} = A_i$. Generally, we terminate in n-EXPTIME. When A_0 satisfies a "non-alternating" property (e.g. when we're only interested in reaching a designated control state), we can restrict Π to only add transitions where Q_n has at most one element, giving $(n-1)$-EXPTIME complexity. In all cases saturation is linear in the size of Σ.

4. Examples of Saturation

As an example of saturation, consider a CPDS with the run

$$\langle p_1, [b]\,[c]\,[d]\rangle \xrightarrow{push^2_a} \langle p_2, [ab]\,[c]\,[d]\rangle \xrightarrow{push_2}$$
$$\langle p_3, [ab]\,[ab]\,[c]\,[d]\rangle \xrightarrow{collapse_2} \langle p_4, [c]\,[d]\rangle \xrightarrow{pop_2} \langle p_5, [d]\rangle .$$

Figure 5 shows the sequence of saturation steps, beginning with an accepting run of $\langle p_5, [d]\rangle$ and finishing with an accepting run of $\langle p_1, [b]\,[c]\,[d]\rangle$. The individual steps are explained below.

Initial Automaton We begin at the top of Figure 5 with a stack automaton containing the transitions $q_{p_5} \xrightarrow{q_1} \emptyset$ and $q_1 \xrightarrow[\emptyset]{d} \emptyset$, which we write $q_{p_5} \xrightarrow[\emptyset]{d} (\emptyset, \emptyset)$. This gives the pictured run over $\langle p_5, [d]\rangle$.

Rule (p_4, c, pop_2, p_5) When the saturation step considers such a pop rule, it adds $q_{p_4} \xrightarrow[\emptyset]{c} (\emptyset, \{q_{p_5}\})$. We add such a transition because we only require the top order-1 stack (removed by pop_2) to have the top character c (hence $\emptyset$ is the next order-1 label), and after the pop_2 the remaining stack needs to be accepted from q_{p_5} (hence $\{q_{p_5}\}$ is the next order-2 label). This new transition allows us to construct the next run over $\langle p_4, [c]\,[d]\rangle$ in Figure 5.

Rule $(p_3, a, collapse_2, p_4)$ Similarly to the pop rule above, the saturation step adds the transition $q_{p_3} \xrightarrow[\{q_{p_4}\}]{a} (\emptyset, \emptyset)$. The addition of such a transition allows us to construct the pictured run over $\langle p_3, [ab]\,[ab]\,[c]\,[d]\rangle$ (collapse links omitted), recalling that $\emptyset \xrightarrow{\emptyset} \emptyset$, $\emptyset \xrightarrow[\emptyset]{a} \emptyset$ and $\emptyset \xrightarrow[\emptyset]{b} \emptyset$ transitions are always possible due to the empty initial set. Note that the labelling of $\{q_{p_4}\}$ in the run comes from the collapse link on the topmost a character on the stack.

Rule $(p_2, a, push_2, p_3)$ Consider the run from q_{p_3} in Figure 5. The initial transition of the run accepting the first order-1 stack is $q_{p_3} \xrightarrow[\{q_{p_4}\}]{a} (\emptyset, \emptyset)$. We also have $\emptyset \xrightarrow{\emptyset} \emptyset$ (trivially) accepting the second order-1 stack. Any $push_2$ predecessor of this stack must have a top order-1 stack that could have appeared twice at the top of the stack from q_{p_3}. Thus, the saturation step makes the intersection of the initial order-1 transitions of first two order-1 stacks. This results in the transition $q_{p_2} \xrightarrow[\{q_{p_4}\} \cup \emptyset]{a} (\emptyset \cup \emptyset, \emptyset)$, which is used to form the shown run over $\langle p_2, [ab]\,[c]\,[d]\rangle$ (collapse links omitted).

Rule $\left(p_1, b, push^2_a, p_2\right)$ The run from q_{p_2} in Figure 5 begins with $q_{p_2} \xrightarrow[\{q_{p_4}\}]{a} (\emptyset, \emptyset)$ and $\emptyset \xrightarrow[\emptyset]{b} \emptyset$. Note that the $push^2_a$ gives a stack with ab on top. Moreover, the collapse link on a should point to the order-1 stack just below the current top one. Since the transition from q_{p_2} requires that the linked-to stack is accepted from q_{p_4}, we need this requirement in the preceding stack (accepted from q_{p_1} and without the a on top). Thus, we move the target of the collapse link into the order-2 destination of the new transition. That is, the saturation step for $push^2_a$ rules creates $q_{p_1} \xrightarrow[\emptyset]{b} (\emptyset, \emptyset \cup \{q_{p_4}\})$. This can be used to construct an accepting run over $\langle p_1, [b]\,[c]\,[d]\rangle$.

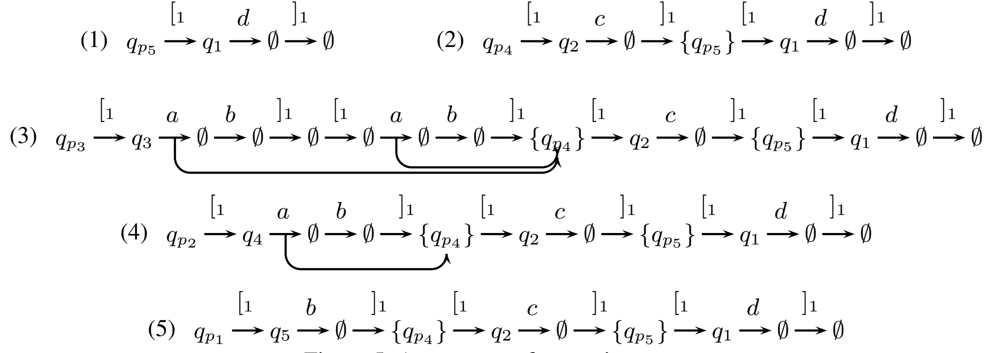

Figure 5: A sequence of saturation steps.

5. Initial Forward Analysis

In this section we distinguish an *error state* p_{err} and we are interested only in whether $\mathcal{C}$ can reach a configuration of the form $\langle p_{err}, w\rangle$ (hence our A_0 is "non-alternating"). This suffices to capture the same safety (reachability) properties of recursion schemes as TRecS. We fix a stack-automaton $\boldsymbol{\mathcal{E}}$ recognising all error configurations (those with the state p_{err}). We write $Post^*_{\mathcal{C}}$ for the set of configurations reachable by $\mathcal{C}$ from the initial configuration. This set cannot be represented precisely by a stack automaton [5] (for instance using $push_2$, we can create $[[a^n]_1[a^n]_1]_2$ from $[[a^n]_1]_2$ for any $n \geq 0$). We summarise our approach then give details in Sections 5.1, 5.2 and 5.3.

It is generally completely impractical to compute $Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}})$ in full (most non-trivial examples considered in our experiments would time-out). For our saturation algorithm to be usable in practice, it is therefore essential that the search space is restricted, which we achieve by means of an initial forward analysis of the CPDS. Ideally we would compute only $Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}}) \cap Post^*_{\mathcal{C}}$. Since this cannot be represented by an automaton, we instead compute a sufficient approximation T (ideally a *strict* subset of $Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}})$) where:

$$Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}}) \ \cap\ Post^*_{\mathcal{C}} \ \subseteq\ T \ \subseteq\ Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}}).$$

The initial configuration will belong to T iff it can reach a configuration recognised by $\boldsymbol{\mathcal{E}}$. Computing such a T is much more feasible.

We first compute an over-approximation of $Post^*_{\mathcal{C}}$. For this we use a *summary algorithm* [34] (that happens to be precise at order-1) from which we extract an over-approximation of the set of CPDS rules that may be used on a run to p_{err}. Let $\mathcal{C}'$ be the (smaller) CPDS containing only these rules. That is, we remove all rules that we know cannot appear on a run to p_{err}. We could thus take $T = Pre^*_{\mathcal{C}'}(\boldsymbol{\mathcal{E}})$ (computable by saturation for $\mathcal{C}'$) since it satisfies the conditions above. This is what we meant by *'pruning'* the CPDS (1a in the list on page 2)

However, we further improve performance by computing an even smaller T (1b in the list on page 2). We extract contextual information from our over-approximation of $Post^*_{\mathcal{C}}$ about how pops and collapses might be used during a run to p_{err}. Our $\mathcal{C}'$ is then restricted to a model $\mathcal{C}''$ that 'guards' its rules by these contextual constraints. Taking $T = Pre^*_{\mathcal{C}''}(\boldsymbol{\mathcal{E}})$ we have a T smaller than $Pre^*_{\mathcal{C}'}(\boldsymbol{\mathcal{E}})$, but still satisfying our sufficient conditions. In fact, $\mathcal{C}''$ will be a *'guarded CPDS'* (defined in the next subsection). We cannot compute $Pre^*_{\mathcal{C}''}(\boldsymbol{\mathcal{E}})$ precisely for a guarded CPDS, but we can adjust saturation to compute T such that $Pre^*_{\mathcal{C}''}(\boldsymbol{\mathcal{E}}) \ \subseteq\ T \ \subseteq\ Pre^*_{\mathcal{C}'}(\boldsymbol{\mathcal{E}})$. This set will thus also satisfy our sufficient conditions.

5.1 Guarded Destruction

An *order-n guarded CPDS (n-GCPDS)* is an n-CPDS where conventional pop_k and $collapse_k$ operations are replaced by *guarded operations* of the form pop^S_k and $collapse^S_k$ where $S \subseteq \Sigma$. These operations may only be fired if the resulting stack has a member of S on top. That is, for $o \in \{collapse_k, pop_k \mid 1 \leq k \leq n\}$:

$$o^S(u) := \begin{cases} o(u) & \text{if } o(u) \text{ defined and } top_1(o(u)) \in S \\ \text{undefined} & \text{otherwise .} \end{cases}$$

Note, we do not guard the other stack operations since these themselves guarantee the symbol on top of the new stack (e.g. when a transition $(p, a, push_2, p')$ fires it must always result in a stack with a on top, and $(p, a, push^b_k, p')$ produces a stack with b on top).

Given a GCPDS $\mathcal{C}$, we write $\mathsf{Triv}(\mathcal{C})$ for the ordinary CPDS that is the *trivialisation* of $\mathcal{C}$, obtained by replacing each pop^S_k (resp. $collapse^S_k$) in the rules of $\mathcal{C}$ with pop_k (resp. $collapse_k$).

We modify the saturation algorithm to use 'guarded' saturation steps for pop and collapse rules. Other saturation steps are unchanged. Non-trivial guards reduce the size of the stack-automaton constructed by avoiding certain additions that are only relevant for unreachable (and hence uninteresting) configurations in the pre-image. This thus improves performance.

1. when $o = pop^S_k$, for each $q_{p'} \xrightarrow{q_k} (Q_{k+1}, \ldots, Q_n)$ in A such that there is a transition of the form $q_k \xrightarrow[_]{b} (_, \ldots, _)$ in A such that $b \in S$, add $q_p \xrightarrow[\emptyset]{a} (\emptyset, \ldots, \emptyset, \{q_k\}, Q_{k+1}, \ldots, Q_n)$ to A',

3. when $o = collapse^S_k$, for each $q_{p'} \xrightarrow{q_k} (Q_{k+1}, \ldots, Q_n)$ in A such that there is a transition of the form $q_k \xrightarrow[_]{b} (_, \ldots, _)$ in A with $b \in S$, add $q_p \xrightarrow[\{q_k\}]{a} (\emptyset, \ldots, \emptyset, Q_{k+1}, \ldots, Q_n)$ to A'.

E.g., suppose that an ordinary (non-guarded) 2-CPDS has rules $(p_1, c, collapse_2, p)$ and $(p_2, d, collapse_2, p')$. The *original* saturation algorithm would process these rules to add the transitions:

$$q_{p_1} \xrightarrow[\{q_p\}]{c} (\emptyset, \emptyset) \qquad \text{and} \qquad q_{p_2} \xrightarrow[\{q_{p'}\}]{d} (\emptyset, \emptyset)$$

Now suppose that the saturation algorithm has produced two transitions of the form $q_p \xrightarrow[_]{a} (_, _)$ and $q_{p'} \xrightarrow[_]{b} (_, _)$. If a GCPDS had, for example, the rules $(p_1, c, collapse^{\{a\}}_2, p)$ and $(p_2, d, collapse^{\{b\}}_2, p')$, then these same two transitions would be added by the modified saturation algorithm. On the other hand, the rules $(p_1, c, collapse^{\{a\}}_2, p)$ and $(p_2, d, collapse^{\{a\}}_2, p')$ would only result in the first of the two transitions being added.

Lemma 5.1. *The revised saturation algorithm applied to $\boldsymbol{\mathcal{E}}$ (for a GCPDS $\mathcal{C}$) gives a stack automaton recognising T such that $Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}}) \ \subseteq\ T \ \subseteq\ Pre^*_{\mathsf{Triv}(\mathcal{C})}(\boldsymbol{\mathcal{E}})$*

Remark 5.1. *The algorithm may result in a stack-automaton recognising configurations that do not belong to $Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}})$ (although still in $Pre^*_{\mathsf{Triv}(\mathcal{C})}(\boldsymbol{\mathcal{E}})$). This is because a state q_k having a transition $q_k \xrightarrow[_]{b} (_, \ldots, _)$ may also have another transition*

$q_k \xrightarrow{b'} (_, \ldots, _)$ *with* $b \neq b'$ *(and so it might recognise a stack from which a* pop_k*, say, guarded by* b *cannot be performed).*

Remark 5.2. *The above modification to the naive saturation algorithm can also be easily incorporated into the efficient fixed point algorithm described in Section 7.*

5.2 Approximate Reachability Graphs

We now give an overview of the summary algorithm used to obtain an over-approximation of $Post^*_{\mathcal{C}}$ and thus compute the GCPDS $\mathcal{C}''$ mentioned previously. We refer the reader to the long version of this paper for details, including a formal account of the invariants on the graph maintained by the algorithm. For simplicity, we assume that a stack symbol uniquely determines the order of any link that it emits (which is the case for a CPDS obtained from a HORS).

An *approximate reachability graph* for $\mathcal{C}$ is a structure (H, E) describing an over-approximation of the reachable configurations of $\mathcal{C}$. The set of nodes of the graph H consists of *heads* of the CPDS, where a head is a pair $(p, a) \in \mathcal{P} \times \Sigma$ and describes configurations of the form $\langle p, u \rangle$ where $top_1(u) = a$. The set E contains directed edges $((p, a), r, (p', a'))$ labelled by rules of $\mathcal{C}$. Such edges over-approximate the transitions that $\mathcal{C}$ might make using a rule r from a configuration described by (p, a) to one described by (p', a'). For example, suppose that $\mathcal{C}$ is order-2 and has, amongst others, the rules $r_1 := (p_1, b, push_2, p_2)$, $r_2 := (p_2, b, push^2_c, p_3)$ and $r_3 := (p_3, c, pop_1, p_4)$ so that it can perform transitions:

$$\left\langle p_1, \left[\left[\begin{array}{c} b \\ a \end{array}\right]\right]\right\rangle \xrightarrow{r_1} \left\langle p_2, \left[\left[\begin{array}{c} b \\ a \end{array}\right] \left[\begin{array}{c} b \\ a \end{array}\right]\right]\right\rangle$$

$$\xrightarrow{r_2} \left\langle p_3, \left[\left[\begin{array}{c} c \\ b \\ a \end{array}\right] \left[\begin{array}{c} b \\ a \end{array}\right]\right]\right\rangle \xrightarrow{r_3} \left\langle p_4, \left[\left[\begin{array}{c} b \\ a \end{array}\right] \left[\begin{array}{c} b \\ a \end{array}\right]\right]\right\rangle$$

where the first configuration mentioned here is reachable. We should then have edges $((p_1, b), r_1, (p_2, b))$, $((p_2, b), r_2, (p_3, c))$ and $((p_3, c), r_3, (p_4, b))$ in E. We denote the configurations above C_1, C_2, C_3 and C_4 respectively, with respective stacks s_1, s_2, s_3, s_4.

Such a graph can be computed using an *approximate summary algorithm*, which builds up an object (H, E, B, U) consisting of an approximate reachability graph together with two additional components. B is a map assigning each head h in the graph a set $B(h)$ of *stack descriptors*, which are $(n+1)$-tuples $(h_n, \ldots, h_1, h_c)$ of heads. In the following, we refer to h_k as the order-k component and h_c the collapse component. Roughly speaking, h_k describes at which head the new top_k-stack resulting from a pop_k operation (applied to a configuration with head h) may have been created, and h_c does likewise for a *collapse* operation. (We will use $\perp$ in place of a head to indicate when pop_k or *collapse* is undefined.)

Consider $C_3 = \langle p_3, s_3 \rangle$ from the example above. This has control-state p_3 and top stack symbol c and so is associated with the head (p_3, c). Thus $B((p_3, c))$ should contain the stack-descriptor $((p_1, b), (p_2, b), (p_1, b))$, which describes s_3. The first (order-2) component is because $top_2(s_3)$ was created by a $push_2$ operation from a configuration with head (p_1, b). The second (order-1) component is because the top symbol was created via an order-1 push from (p_2, b). Finally, the order-2 link from the top of s_3 points to a stack occurring on top of a configuration at the head (p_1, b), giving rise to the final (collapse) component describing the collapse link.

Tracking this information allows the summary algorithm to process the rule r_3 to obtain a description of C_4 from the description of C_3. Since this rule performs a pop_1, it can look at the order-1 component of the stack descriptor to see the head (p_2, b), telling us that pop_1 results in b being on top of the stack. Since the rule r_3 moves into control-state p_4, this tells us that the new head should be (p_4, b). It also tells us that certain pieces of information in $B((p_2, b))$ are relevant to the description of $top_2(s_4)$ contained in $B((p_4, b))$. First remark that this situation only occurs for the pop_k and $collapse_k$ operations. To keep track of these correlations, we use the component U of the graph.

The component U is a set of *approximate higher-order summary edges*. A summary edge describes how information contained in stack descriptors should be shared between heads. An *order-k summary edge* from a head h to a head h' is a triple of the form $(h, (h'_n, \ldots, h'_{k+1}), h')$ where each h'_i is a head. Such a summary edge is added when processing either a pop_k or a $collapse_k$ operation on an order-k link. Intuitively such a summary edge means that if $(h_n, \ldots, h_{k+1}, h_k, \ldots, h_1, h_c) \in B(h)$, then we should also have $(h'_n, \ldots, h'_{k+1}, h_k, \ldots, h_1, h_c) \in B(h')$. To continue our example, the r_3 rule (which performs a pop_1 operation) from C_3 to C_4 means U should contain an order-1 summary edge $((p_2, b), ((p_1, b)), (p_4, b))$. Since pop_1 is an order-1 operation, we have $pop_2(s_3) = pop_2(s_4)$. Hence (p_1, b) (the order-2 component of the stack descriptor for s_3) should also be the first component of a stack descriptor for s_4. However, since $top_1(s_4)$ was created at a configuration with head (p_2, b), the order-1 and collapse components of such a stack descriptor for s_4 should be inherited from a stack descriptor in $B((p_2, b))$. In general if we go from a configuration (p, s) with h to a configuration (p', s') with head h' by the pop_k operation or $collapse_k$ on an order-k link, we have that $pop_{k+1}(s) = pop_{k+1}(s')$ and hence we have a summary edge $(h, (h'_n, \ldots, h'_{k+1}), h')$

The construction of the approximate reachability graph is described in algorithms 7, 8, 9 and 10. The main work is done in the function $\mathrm{ProcessHeadWithDescriptor}$. In particular, this is where summary edges are added for the pop_k and $collapse_k$ operations. A fully worked example is given in the long version.

5.3 Extracting the Guarded CPDA

Let $\mathcal{G} = (H, E)$ be an approximate reachability graph for $\mathcal{C}$. Let $\mathsf{Heads}(\boldsymbol{\mathcal{E}})$ be the set of heads of error configurations, *i.e.* $\mathsf{Heads}(\boldsymbol{\mathcal{E}}) := \{(p_{err}, a) \mid a \in \Sigma\}$. We do a simple backwards reachability computation on the finite graph $\mathcal{G}$ to compute the set $\mathsf{BackRules}(\mathcal{G})$, which is defined to be the smallest set satisfying:

$$\mathsf{BackRules}(\mathcal{G}) = \left\{ e \in E \;\middle|\; \begin{array}{l} e = (h, r, h') \in E \text{ for some} \\ h' \in \mathsf{Heads}(\boldsymbol{\mathcal{E}}) \end{array} \right\} \cup \left\{ e \in E \;\middle|\; \begin{array}{l} e = (h, r, h') \in E \text{ for some} \\ (h', _, _) \in \mathsf{BackRules}(\mathcal{G}) \end{array} \right\}$$

The CPDS rules occurring in the triples in $\mathsf{BackRules}(\mathcal{G})$ can be used to define a pruned CPDS $\mathcal{C}'$ that reaches an error state if and only if the original also does. However, the approximate reachability graph provides enough information to construct a guarded CPDS $\mathcal{C}''$ whose guards are non-trivial. It should be clear that the following set $\mathsf{BackRulesG}(\mathcal{G})$ of *guarded* rules can be computed:

$$\left\{ (p, a, o', p') \;\middle|\; \begin{array}{l} (_, (p, a, o, p'), _) \in \mathsf{BackRules}(\mathcal{G}) \text{ and} \\ o' = \begin{cases} o^S & \text{if } o \text{ is a pop or a collapse and } S \\ & = \left\{ b \;\middle|\; \begin{array}{l} ((p, a), r, (p', b)) \\ \in E \end{array} \right\} \\ & \text{with } r = (p, a, o, p') \\ o & \text{if } o \text{ is a rewrite or push} \end{cases} \end{array} \right\}$$

These rules define a GCPDS on which C-SHORe finally performs saturation.

Lemma 5.2. *The GCPDS* $\mathcal{C}''$ *defined using* $\mathsf{BackRulesG}(\mathcal{G})$ *satisfies:* $Post^*_{\mathcal{C}} \cap Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}}) \subseteq Pre^*_{\mathcal{C}''}(\boldsymbol{\mathcal{E}}) \subseteq Pre^*_{\mathcal{C}}(\boldsymbol{\mathcal{E}})$

Algorithm 1 The Approximate Summary Algorithm

Require: An n-CPDS with rules $\mathcal{R}$ and heads $\mathcal{P} \times \Sigma$ and initial configuration $\langle p_0, [\cdots [a_0]_1 \cdots]_n \rangle$
Ensure: The creation of a structure (H, E, B, U) where (H, E) is an approximate reachability graph and U is a set of approximate higher-order summary edges.

Set $H := \{(p_0, a_0)\}$ and set E, B and U to be empty
Call AddStackDescriptor($(p_0, a_0), (\perp, \ldots, \perp, \perp)$)
return Done, (H, E, B, U) will now be as required

Algorithm 2 AddStackDescriptor($h, (h_n, \ldots, h_1, h_c)$)

Require: A head $h \in H$ and a stack descriptor $(h_n, \ldots, h_1, h_c)$
Ensure: $(h_n, \ldots, h_1, h_c) \in B(h)$ and all additions to $B(h')$ for all $h' \in H$ needed to respect summary edges are made.

if $(h_n, \ldots, h_1, h_c) \in B(h)$ **then**
  return Done (Nothing to do)
Add $(h_n, \ldots, h_1, h_c)$ to $B(h)$
Call ProcessHeadWithDescriptor($h, (h_n, \ldots, h_1, h_c)$)
for $h' \in H$ such that $(h, (h'_n, \ldots, h'_{k+1}), h') \in U$ **do**
  Call AddStackDescriptor($h', (h'_n, \ldots, h'_{k+1}, h_k, \ldots, h_1, h_c)$)
return Done

Algorithm 3 ProcessHeadWithDescriptor($h, (h_n, \ldots, h_1, h_c)$)

Require: A head $h := (p, a) \in H$ and a stack descriptor $(h_n, \ldots, h_1, h_c) \in B(h)$
Ensure: All necessary modifications to the graph are made so that it is consistent with $(h_n, \ldots, h_1, h_c) \in B(h)$. In particular this is the procedure that processes the CPDS rules from h (with respect to a stack described by h and the stack descriptor).

for o and p' such that $r = (p, a, o, p') \in \mathcal{R}$ **do**
  if $o = rew_b$ **then**
    Add (p', b) to H and $((p, a), r, (p', b))$ to E
    Call AddStackDescriptor($(p', b), (h_n, \ldots, h_1, h_c)$)
  else if $o = push^k_b$ **then**
    Add (p', b) to H and $((p, a), r, (p', b))$ to E
    Call AddStackDescriptor($(p', b), (h_n, \ldots, h_2, (p, a), h_k)$)
  else if $o = push_k$ **then**
    Add (p', a) to H and $((p, a), r, (p', a))$ to E
    Call AddStackDescriptor($(p', a), (h_n, \ldots, h_{k+1}, (p, a),$
      $h_{k-1}, \ldots, h_1, h_c)$)
  else if $o = pop_k$ with $h_k = (p_k, a_k)$ where $a_k \neq \perp$ **then**
    Add (p', a_k) to H and $((p, a), r, (p', a_k))$ to E
    Call AddSummary($(p_k, a_k), (h_n, \ldots, h_{k+1}), (p', a_k)$)
  else if $o = collapse_k$, $h_c = (p_c, a_c)$ and $a_c \neq \perp$ **then**
    Add (p_c, a_c) to H and $((p, a), r, (p', a_c))$ to E
    Call AddSummary($(p_c, a_c), (h_n, \ldots, h_{k+1}), (p', a_c)$)
return Done

Algorithm 4 AddSummary($h, (h'_n, \ldots, h_{k+1}), h'$)

Require: An approximate higher-order summary edge $(h, (h'_n, \ldots, h_{k+1}), h')$
Ensure: $(h, (h'_n, \ldots, h'_{k+1}), h') \in U$ and that all necessary stack descriptors are added to the appropriate $B(h'')$ for $h'' \in H$ so that all summary edges (including the new one) are respected.

if $(h, (h'_n, \ldots, h'_{k+1}), h') \in U$ **then**
  return Done (Nothing to do)
Add $(h, (h'_n, \ldots, h'_{k+1}), h')$ to U
for $(h_n, \ldots, h_{k+1}, h_k, \ldots, h_1, h_c) \in B(h)$ **do**
  AddStackDescriptor($h', (h'_n, \ldots, h'_{k+1}, h_k, \ldots, h_1, h_c)$)
return Done

6. Counter Example Generation

In this section, we describe an algorithm that given a CPDS $\mathcal{C}$ and a stack automaton A_0 such that a configuration $\langle p, w \rangle$ of $\mathcal{C}$ belongs $Pre^*_{\mathcal{C}}(A_0)$, constructs a sequence of rules of $\mathcal{C}$ which when applied from $\langle p, w \rangle$ leads to a configuration in $\mathcal{L}(A_0)$. In practice, we use the algorithm with A_0 accepting the set of all configurations starting with some error state p_{err}. The output is a counter-example showing how the CPDS can reach this error state.

The algorithm itself is a natural one and the full details are given in the full version of this paper. We describe it informally here by means of the example in Figure 5, described in Section 4.

To construct a trace from $\langle p_1, [b]\,[c]\,[d] \rangle$ to $\langle p_5, [d] \rangle$ we first note that, when adding the initial transition of the pictured run from q_{p_1}, the saturation step marked that the transition was added due to the rule $(p_1, b, push^2_a, p_2)$. If we apply this rule to $\langle p_1, [b]\,[c]\,[d] \rangle$ we obtain $\langle p_2, [ab]\,[c]\,[d] \rangle$ (collapse links omitted). Furthermore, the justifications added during the saturation step tell us which transitions to use to construct the pictured run from q_{p_2}. Hence, we have completed the first step of counter example extraction and moved one step closer to the target configuration. To continue, we consider the initial transition of the run from q_{p_2}. Again, the justifications added during saturation tell us which CPDS rule to apply and which stack automaton transitions to use to build an accepting run of the next configuration. Thus, we follow the justifications back to a run of A_0, constructing a complete trace on the way.

The main technical difficulty lies in proving that the reasoning outlined above leads to a terminating algorithm. For example, we need to prove that following the justifications does not result us following a loop indefinitely. Since the stack may shrink and grow during a run, this is a non-trivial property. To prove it, we require a subtle relation on runs over higher-order collapsible stacks.

6.1 A Well-Founded Relation on Stack Automaton Runs

We aim to define a well-founded relation over runs of the stack automaton A constructed by saturation from $\mathcal{C}$ and A_0. To do this we represent a run over a stack as another stack of (sets of) transitions of A. This can be obtained by replacing each instance of a stack character with the set of order-1 transitions that read it. This is formally defined in the full version of the paper and described by example here. Consider the run over $[[b]\,[c]\,[d]]$ from q_{p_1} in Figure 5. We can represent this run as the stack $[[\{t_1\}]\,[\{t_2\}]\,[\{t_3\}]]$ where $t_1 = q_5 \xrightarrow[\emptyset]{b} \emptyset$, $t_2 = q_2 \xrightarrow[\emptyset]{c} \emptyset$ and $t_3 = q_1 \xrightarrow[\emptyset]{d} \emptyset$. Note that since q_5 uniquely labels the order-2 transition $q_{p_1} \xrightarrow{q_5} \{q_{p_4}\}$ (and similarly for the transitions from q_{p_4} and q_{p_5}) we do not need to explicitly store these transitions in our stack representation of runs.

Using this representation, we can define by induction a relation $\hookrightarrow_k$ on the order-k runs of A. Note that this is not an order relation as it is not always transitive. There are several cases to $\hookrightarrow_k$.

1. For $k = 1$, we say $w' \hookrightarrow_1 w$ if for some $i \geq 0$, w contains strictly fewer transitions in Δ_1 justified at step i than w' and that for all $j > i$ they both contain the same number of transitions in Δ_1 justified at step j.
2. For $k > 1$, we say $u = [u_\ell \ldots u_1]_k \hookrightarrow_k v = [v_{\ell'} \ldots v_1]_k$ if
   (a) $\ell' < \ell$ and $u_i = v_i$ for $i \in [1, \ell' - 1]$ and either $u_{\ell'} = v_{\ell'}$ or $u_{\ell'} \hookrightarrow_{k-1} v_{\ell'}$, or
   (b) $\ell' \geq \ell$ and $u_i = v_i$ for $i \in [1, \ell - 1]$ and $u_\ell \hookrightarrow_{k-1} v_i$ for all $i \in [\ell, \ell']$.

Lemma 6.1. *For all $k \in [1, n]$, the relation $\hookrightarrow_k$ is well-founded. Namely there is no infinite sequence $w_0 \hookrightarrow_k w_1 \hookrightarrow_k w_2 \hookrightarrow_k \cdots$.*

It is possible to show that by following the justifications, from stack w to a w', we always have $w \hookrightarrow_n w'$. Since this relation is well-founded, the witness generation algorithm always terminates.

7. Efficient Fixed Point Computation

We introduce an efficient method of computing the fixed point in Section 3, inspired by Schwoon *et al.*'s algorithm for alternating (order-1) pushdown systems [36]. Rather than checking all CPDS rules at each iteration, we fully process *all* consequences of each new transition at once. New transitions are kept in a set Δ_{new} (implemented as a stack), processed, then moved to a set Δ_{done}, which forms the transition relation of the final stack automaton. We assume w.l.o.g. that a character's link order is determined by the character. This is true of all CPDSs obtained from HORSs.

In most cases, new transitions only depend on a single existing transition, hence processing the consequences of a new transition is straightforward. The key difficulty is the push rules, which depend on *sets* of existing transitions. Given a rule $(p, a, push_k, p')$, processing $t = q_{p'} \xrightarrow[Q_{col}]{a} (Q_1, \ldots, Q_k, \ldots, Q_n)$ 'once and once only' must somehow include adding a new transition whenever there is a set of transitions of the form $Q_k \xrightarrow[Q'_{col}]{a} (Q'_1, \ldots, Q'_k)$ in A_i either now *or in the future*. When t is processed, we therefore create a *trip-wire*, consisting of a *source* and *target*. A *target* collects transitions from a given set of states (such as Q_k above), whilst a *source* describes how such a collection could be used to form a new transition according to a *push* saturation step.

Definition 7.1. *An* order-k source *for $k \geq 1$ is defined as a tuple (q_k, q_{k-1}, a, Q_k) in $\mathbb{Q}_k \times \mathbb{Q}^{\perp}_{k-1} \times \Sigma \times 2^{\mathbb{Q}_k}$ where $\mathbb{Q}^{\perp}_0 := \{\perp\}$ and $\mathbb{Q}^{\perp}_i = \mathbb{Q}_i \cup \{\perp\}$ for $i \geq 1$. An* order-k target *is a tuple*

$$(Q_k, Q^C_k, Q_{lbl}, Q'_k) \in 2^{\mathbb{Q}_k} \times 2^{\mathbb{Q}_k} \times 2^{\mathbb{Q}_{k-1}} \times 2^{\mathbb{Q}_k}$$

if $k \geq 2$, and if $k = 1$

$$(Q_1, Q^C_1, a, Q_{col}, Q'_1) \in \bigcup_{k'=2}^{n} \left(2^{\mathbb{Q}_1} \times 2^{\mathbb{Q}_1} \times \Sigma \times 2^{\mathbb{Q}_{k'}} \times 2^{\mathbb{Q}_1}\right) .$$

The set Q^C_k is a *countdown* containing states in Q_k still awaiting a transition. We always have $Q^C_k \subseteq Q_k$ and $\left(Q_k \setminus Q^C_k\right) \xrightarrow{Q_{lbl}} Q'_k$. Likewise, an order-1 target $(Q_1, Q^C_1, a, Q_{col}, Q_1)$ will satisfy $(Q_1 \setminus Q^C_1) \xrightarrow[Q_{col}]{a} Q'_1$. A target is *complete* if $Q^C_k = \emptyset$ or $Q^C_1 = \emptyset$.

A *trip-wire* of order-k is an order-k source-target pair of the form $((_, _, _, Q_k), (Q_k, _, _, _))$ when $k \geq 2$ or $((_, _, a, Q_k), (Q_k, _, a, _, _))$ when $k = 1$. When the target in a trip-wire is *complete*, the action specified by its source is triggered, which we now sketch.

An order-k *source* for $k \geq 2$ describes how an order-$(k-1)$ source should be created from a complete target, propagating the computation to the level below, and an order-1 source describes how a new long-form transition should be created from a complete target. That is, when we have $(q_k, _, a, Q_k)$ (we hide the second component for simplicity of description) and $(Q_k, \emptyset, Q_{lbl}, Q'_k)$ this means we've found a set of transitions witnessing $Q_k \xrightarrow{Q_{lbl}} Q'_k$ and should now look for transitions from Q_{lbl}. Hence the algorithm creates a new source and target for the order-$(k-1)$ state-set Q_{lbl}. When this process reaches order-1, a new transition is created. This results in the construction of the t' from a *push* saturation step.

Algorithm 5 gives the main loop and introduces the global sets Δ_{done} and Δ_{new}, and two arrays $\mathcal{U}_{src}[k]$ and $\mathcal{U}_{targ}[k]$ containing sources and targets for each order. Omitted are loops processing pop_n and $collapse_n$ rules like the naive algorithm. Algorithm 6 gives the main steps processing a new transition. We present only two CPDS rule cases here. In most cases a new transition is created, however, for *push* rules we create a trip-wire. Remaining algorithms, definitions, justification handling, and proofs are given in the long version of this paper. We describe some informally below.

In create_trip_wire we create a trip-wire with a new target $(Q_k, Q_k, \emptyset, \emptyset)$. This is added using an add_target procedure which also checks Δ_{done} to create further targets. E.g., a new target $\left(Q_k, Q^C_k, Q_{lbl}, Q'_k\right)$ combines with an existing $q_k \xrightarrow{q_{k-1}} Q''_k$ to create a new target $\left(Q, Q^C_k \setminus \{q_k\}, Q_{lbl} \cup \{q_{k-1}\}, Q'_k \cup Q''_k\right)$. (This step corrects a bug in the algorithm of Schwoon *et al.*) Similarly update_trip_wires updates existing targets by new transitions. In all cases, when a source and matching complete target are created, we perform the propagations described above.

Proposition 7.1. *Given a CPDS $\mathcal{C}$ and stack automaton A_0, let A be the result of Algorithm 5. We have $\mathcal{L}(A) = Pre^*_{\mathcal{C}}(A_0)$.*

8. Experimental Results

We compared C-SHORe with the current state-of-the-art verification tools for higher-order recursion schemes (HORS): TRecS [23], GTRecS2 [26] (the successor of [25]), and TravMC [28]. Benchmarks are from the TRecS and TravMC benchmark suites, plus several larger examples provided by Kobayashi. The majority of the TravMC benchmarks were translated into HORS from an extended formalism, HORS with Case statements (HORSC), using a script by Kobayashi. For fairness, all tools in our experiments took a pure HORS as input. However, the authors of TravMC report that TravMC performs faster on the original HORSC examples than on their HORS translations.

In all cases, the benchmarks consist of a HORS (generating a computation tree) and a property automaton. In the case of C-SHORe, the property automaton is a regular automaton describing branches of the generated tree that are considered errors. Thus, following the intuition in Section 2, we can construct a reachability query over a CPDS, where the reachability of a control state p_{err} indicates an erroneous branch (see [10] for more details). All other tools check co-reachability properties of HORS and thus the property automaton describes only valid branches of the computation tree. In all cases, it was straightforward to translate between the co-reachability and reachability properties.

The experiments were run on a Dell Latitude e6320 laptop with 4Gb of RAM and four 2.7GHz Intel i7-2620M cores. We ran C-SHORe on OpenJDK 7.0 with IcedTea7 replacing binary plugs, using the argument "-Xmx" to limit RAM usage to 2.5Gb. As advised by the TravMC developers, we ran TravMC on the Mono JIT Compiler version 3.0.3 with no command line arguments. Finally TRecS (version 1.34) and GTRecS2 (version 3.17) were compiled with the OCaml version 4.00.1 compilers. On negative examples, GTRecS2 was run with its `-neg` argument. We used the "ulimit" command to limit memory usage to 2.5Gb and set a CPU timeout of 600 seconds (per benchmark). The given runtimes were reported by the respective tools and are the means of three separate runs on each example. Note that C-SHORe was run until the automaton was completely saturated.

Table 1 shows trials where at least one tool took over 1s. This is because virtual machine "warm-up" and HORS to CPDS conversion can skew the results on small benchmarks. Full results are in the full paper. Examples violating their property are marked "(bug)". The order (Ord) and size (Sz) of the schemes were reported by TRecS. We show reported times in seconds for TRecS (T), GTRecS2 (G), TravMC (TMC) and C-SHORe (C) where "—" means analysis failed. For C-SHORe, we report the times for HORS to CPDS translation (Ctran), CPDS analysis (Ccpds), and building the approximation graph (Capprox). Capprox is part of Ccpds, and the full time (C) is the sum of Ctran and Ccpds.

Of 26 benchmarks, C-SHORe performed best on 5 examples. In 6 cases, C-SHORe was the slowest. In particular, C-SHORe does not perform well on `exp4-1` and `exp4-5`. These belong to a class of benchmarks that stress higher-order model-checkers and indi-

Algorithm 5 Computing $Pre^*_C(A_0)$

Let $\Delta_{done} = \emptyset$, $\Delta_{new} = \bigcup_{n \geq k \geq 1} \Delta_k$, $\mathcal{U}_{src}[k] = \emptyset$, $\mathcal{U}_{targ}[k] = \{(\emptyset, \emptyset, \emptyset, \emptyset)\}$ for each $n \geq k > 1$ and $\mathcal{U}_{targ}[1] = \{(\emptyset, \emptyset, a, \emptyset, \emptyset) \mid a \in \Sigma\}$.
...
while $\exists t \in \Delta_{new}$ **do**
  update_rules(t); update_trip_wires(t); move t from Δ_{new} to Δ_{done}

Algorithm 6 update_rules(t)

if t is an order-k transition for $2 \leq k \leq n$ of the form $q_{p' Q_n \cdots Q_{k+1}} \longrightarrow Q_k$ **then**
  for $p \in \mathcal{P}$ and $a \in \Sigma$ such that $r := (p, a, pop_k, p') \in \mathcal{R}$ **do**
    add_to_worklist$\left(q_p \xrightarrow[\mathbb{Q}_{col}]{a} (\emptyset, \ldots, \emptyset, \left\{q_{p' \mathbb{Q}_n \cdots \mathbb{Q}_{k+1}}\right\}, \mathbb{Q}_{k+1}, \ldots, \mathbb{Q}_n), r\right)$
  for $p \in \mathcal{P}$ and $a \in \Sigma$ such that $r := (p, a, push_k, p') \in \mathcal{R}$ **do**
    create_trip_wire$\left(q_{p, Q_n, \ldots, Q_{k+1}}, q_{p', Q_n, \ldots, Q_{k+1}, Q_k}, a, Q_k, (r, t)\right)$
...

Benchmark file	Ord	Sz	T	TMC	**G**	**C**	Ctran	Ccpds	Capprox	✓/✗
`order5`	5	52	0.007	0.039	**—**	**0.415**	0.057	0.358	0.205	
`order5-2`	5	40	0.022	0.084	**—**	**0.305**	0.050	0.255	0.157	
`order5-variant`	5	55	0.019	0.039	**1.519**	**0.427**	0.057	0.370	0.177	
`filepath`	2	5956	210.102	—	**—**	**0.397**	0.168	0.229	0.221	✓
`filter-nonzero` (bug)	5	484	0.006	0.115	**0.182**	**1.443**	0.100	1.344	1.006	✗
`filter-nonzero-1`	5	890	0.176	211.907	**—**	**4.492**	0.159	4.332	3.484	
`map-head-filter-1`	3	880	0.141	1.343	**—**	**0.400**	0.119	0.281	0.273	
`map-plusone-1`	5	459	0.030	0.736	**—**	**1.247**	0.119	1.128	0.908	
`map-plusone-2`	5	704	1.358	13.962	**—**	**2.634**	0.142	2.491	2.183	
`exp4-1`	4	31	—	0.047	**0.114**	**—**	0.039	—	0.240	✗
`exp4-5`	4	55	—	—	**0.818**	**—**	0.046	—	2.128	✗
`cfa-life2`	14	7648	—	—	**—**	**—**	0.479	—	—	
`cfa-matrix-1`	8	2944	16.937	—	**—**	**19.230**	0.332	18.898	18.892	
`cfa-psdes`	7	1819	17.654	—	**—**	**1.920**	0.273	1.647	1.640	✓
`dna`	2	411	0.031	0.263	**0.046**	**6.918**	0.175	6.743	6.206	✗
`fibstring`	4	29	—	74.569	**0.114**	**—**	0.042	—	0.256	✗
`fold_fun_list`	7	1346	0.519	—	**—**	**1.356**	0.202	1.154	1.147	
`fold_right`	5	1310	31.624	—	**—**	**1.255**	0.191	1.064	1.043	✓
`jwig-cal_main`	2	7627	0.062	0.052	**0.161**	**3.802**	3.739	0.063	0.057	✗
`l`	3	35	—	15.743	**0.010**	**0.248**	0.042	0.206	0.199	
`search-e-church` (bug)	6	837	0.012	0.258	**—**	**4.741**	0.155	4.586	1.760	
`specialize_cps_coerce1-c`	3	2731	—	—	**—**	**1.131**	0.293	0.838	0.830	✓
`tak` (bug)	8	451	—	3.945	**—**	**41.772**	0.136	41.636	34.855	
`xhtmlf-div-2` (bug)	2	3003	0.234	—	**39.961**	**2.743**	2.303	0.440	0.422	
`xhtmlf-m-church`	2	3027	0.238	—	**8.420**	**2.708**	2.319	0.389	0.382	
`zip`	4	2952	22.251	—	**—**	**3.356**	0.295	3.061	1.609	✓

Table 1: Comparison of model-checking tools. Shown in bold are the two fixed-parameter tractable algorithms, GTRecS2 and C-SHORe.

cate that our tool currently does not always scale well. However, C-SHORe seems to show a more promising capacity to scale on larger HORS produced by tools such as MoChi [27], which are particularly pertinent in that they are generated by an actual software verification tool. We also note that C-SHORe timed out on the fewest examples despite not always terminating in the fastest time.

It is also very important to note that C-SHORe and GTRecS2 are the only implemented fixed-parameter tractable algorithms in the literature for HORS model-checking of which we are aware (both TRecS and TravMC have worst-case run-times non-elementary in the size of the recursion scheme). Moreover, C-SHORe generally performs much better than GTRecS2. Thus not only does C-SHORe's performance seem promising when compared to the competition, there is also theoretical reason to suggest that the approach could in principle be scalable, in contrast to some of the alternatives. Thus initial work justifies further investigation into saturation based algorithms for higher-order model-checking.

Finally, we remark that without the forwards analysis described in Section 5, all shown examples except `filepath` timed out. We also note that we did not implement a naive version of the saturation algorithm, where after each change to the stack automaton, each rule of the CPDS is checked for further updates. However, experience implementing PDSolver [15] (for order-1 pushdown systems) indicates that the naive approach is at least an order of magnitude slower than the techniques [36] we generalised in Section 7.

9. Related Work

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 [8], parity and μ-calculus conditions [15], and concurrent systems [1, 37], as well as weighted pushdown systems [31]. In addition to various implementations, efficient versions of these algorithms have also been developed [12, 36].

The saturation algorithm for CPDS that we introduced in [7], extending and improving [14] (and [5]), follows a number of papers solving parity games on the configuration graphs of higher-order automata [6, 9, 11, 16]. While only handling reachability, saturation lends itself well to implementation. This paper describes such a practical incarnation and a number of significant optimisations, such as using a forwards analysis to guide the backward search.

This latter point is an important way in which C-SHORe differs from previous model-checkers for HORS, which employ intersection types and propagate information purely in a forward direction.

This is related to the fact that the latter accept 'co-reachability properties' (represented by trivial Büchi automata) as input, expressing the complement of properties taken by C-SHORe.

Indeed it would be interesting to investigate in more detail how approximate forward and backward analyses of varying degrees of accuracy could be combined for efficiency. It would also be helpful to more closely analyse the relationship between CPDS and type-based algorithms allowing a transfer of ideas. In any case, this paper shows that saturation-based algorithms for HORS/CPDS perform sufficiently well in practice to warrant further study.

To finish, we briefly mention several approaches to analysing higher-order programs with differing aims to ours. In static analysis, k-CFA [35] and CFA2 [39] perform an over-approximative analysis of higher-order languages with at-most first-order granularity. Similarly Jhala *et al.* use refinement types to analyse OCaml programs by reducing the problem to first-order model-checking, which is thus incomplete [19]. Finally, Hopkins *et al.* have produced tools for equivalence checking fragments of ML and Idealized Algol up to order-3 [17, 18].

10. Conclusion

We have considered the problem of verifying safety properties of a model that can be used to precisely capture control-flow in the presence of higher-order recursion. Whilst previous approaches to such an analysis are based on higher-order recursion schemes and intersection types, our approach is based on automata and saturation techniques previously only applied in practice to the first-order case. At a more conceptual level, our algorithm works by propagating information backwards from error states towards the initial state. Moreover, it combines this with an approximate forward analysis to gather information that guides the backward search. In contrast, the preceding type-based algorithms all work by propagating information purely in a forward direction.

Our preliminary work brings new techniques to the table for tackling a problem, which in contrast to its first-order counterpart, has proven difficult to solve in a scalable manner. Our algorithm has the advantage that it accurately models higher-order recursion whilst also being fixed-parameter tractable, therefore giving a theoretical reason for hope that it could scale. In contrast TRecS and TravMC have worst-case run-times non-elementary in the size of the recursion scheme. Our tool also seems to work significantly better in practice than GTRecS2, the only other HORS model-checker in the literature that does enjoy fixed-parameter tractability.

We therefore believe that a C-SHORe-like approach shows much promise and warrants further investigation.

Acknowledgments

We are extremely to Robin Neatherway and Naoki Kobayashi for help with benchmarking, Łukasz Kaiser for web-hosting, and for discussions with Stefan Schwoon. Supported by Fond. Sci. Math. Paris, AMIS (ANR 2010 JCJC 0203 01 AMIS), FREC (ANR 2010 BLAN 0202 02 FREC), VAPF (Région IdF), and EPSRC (EP/K009907/1).

References

[1] M. F. Atig. Global model checking of ordered multi-pushdown systems. In *FSTTCS*, 2010.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

[3] M. Benois. Parties rationnelles du groupe libre. *Comptes-Rendus de l'Acamdémie des Sciences de Paris, Série A*, 269:1188–1190, 1969.

[4] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, 1997.

[5] A. Bouajjani and A. Meyer. Symbolic Reachability Analysis of Higher-Order Context-Free Processes. In *FSTTCS*, 2004.

[6] C. H. Broadbent, A. Carayol, C.-H. L. Ong, and O. Serre. Recursion schemes and logical reflection. In *LICS*, 2010.

[7] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. A saturation method for collapsible pushdown systems. In *ICALP*, 2012.

[8] T. Cachat. *Games on Pushdown Graphs and Extensions*. PhD thesis, RWTH Aachen, 2003.

[9] T. Cachat. Higher order pushdown automata, the Caucal hierarchy of graphs and parity games. In *ICALP*, 2003.

[10] A. Carayol and O. Serre. Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In *LICS*, 2012.

[11] A. Carayol, M. Hague, A. Meyer, C.-H. L. Ong, and O. Serre. Winning Regions of Higher-Order Pushdown Games. In *LICS*, 2008.

[12] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, 2000.

[13] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *INFINITY*, 1997.

[14] M. Hague and C.-H. L. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. *Logical Methods in Computer Science*, 4(4), 2008.

[15] M. Hague and C.-H. L. Ong. Analysing mu-calculus properties of pushdown systems. In *SPIN*, 2010.

[16] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS*, 2008.

[17] D. Hopkins and C.-H. L. Ong. Homer: A higher-order observational equivalence model checker. In *CAV*, 2009.

[18] D. Hopkins, A. S. Murawski, and C.-H. L. Ong. Hector: An equivalence checker for a higher-order fragment of ml. In *CAV*, 2012.

[19] R. Jhala, R. Majumdar, and A. Rybalchenko. Hmc: Verifying functional programs using abstract interpreters. In *CAV*, 2011.

[20] N. D. Jones and S. S. Muchnick. Even simple programs are hard to analyze. *J. ACM*, 24:338–350, April 1977.

[21] T. Knapik, D. Niwinski, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars and panic automata. In *ICALP*, 2005.

[22] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, 2009.

[23] N. Kobayashi. Model-checking higher-order functions. In *PPDP*, 2009.

[24] N. Kobayashi. Higher-order model checking: From theory to practice. In *LICS*, 2011.

[25] N. Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *FOSSACS*, 2011.

[26] N. Kobayashi. GTRECS2: A model checker for recursion schemes based on games and types. A tool available at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/gtrecs2/`, 2012.

[27] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and cegar for higher-order model checking. In *PLDI*, 2011.

[28] R. P. Neatherway, S. J. Ramsay, and C.-H. L. Ong. A traversal-based algorithm for higher-order model checking. In *ICFP*, 2012.

[29] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, 2006.

[30] C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL*, 2011.

[31] 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.

[32] S. Salvati and I. Walukiewicz. Recursive schemes, krivine machines, and collapsible pushdown automata. In *RP*, 2012.

[33] S. Schwoon. *Model-checking Pushdown Systems*. PhD thesis, Technical University of Munich, 2002.

[34] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[35] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991.

[36] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In *ATVA*, 2006.

[37] D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded java programs. In *SPIN*, 2008.

[38] H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing programs via higher-order model checking. In *APLAS*, 2010.

[39] D. Vardoulakis. *CFA2: Pushdown-Flow Analysis for Higher-Order Languages*. PhD thesis, Northeastern University, Boston, 2012.

[40] D. Vardoulakis and O. Shivers. Pushdown flow analysis of first-class control. In *ICFP*, 2011.