C-SHORe
A Collapsible Approach to Verifying Higher-Order Programs

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

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

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

O. 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

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 and Moped). However, there has recently been much interest in a model called higher-order recursion schemes (HORS) (see e.g., Kobayashi’s TRecS 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. In a bid to improve scalability, a number of other algorithms have subsequently been designed and implemented such as Kobayashi et al.’s GTRecS tool and Neatherway et al.’s TravMC 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 that checks safety properties of (OCaML) programs, and EHMVT Verifier for tree processing programs. Both use a model-checker for recursion schemes as a central component. Similarly, Ramsay and Ong 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 [14], 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) [15] 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 [11] [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 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 [1]. 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 [14]. 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 (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.
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.

```haskell
Main = MakeReport Nil
MakeReport x = if * (Commit x)
  then (AddData x MakeReport)
  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

\[ N \phi x \leftrightarrow t \]

where N is a typed non-terminal with (possibly higher-order) arguments \( \phi \) and \( x \). A term \( N t_x t_\phi \) rewrites to \( t \) with \( t_\phi \) substituted for \( \phi \) 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

\[
\begin{align*}
\text{main} & \leftrightarrow M \text{ nil} \\
M x & \leftrightarrow \text{or} (\text{commit } x) (A x M) \\
A y \phi & \leftrightarrow \text{or} (\phi \text{ error}) (\phi \text{ cons } y)
\end{align*}
\]
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 (\( A \times M \))). 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 (\( A \times nil \)). 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, commit, `error``. This is an error situation because `commit` is being called with an `error` report. In general we can define the regular language \( \mathcal{L}_{err} = \{ \text{commit or `error`} \} \). 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 \((p, w)\), where \( p \) is a control state taken from a finite set \( P \), 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 \( \Sigma \). An order-2 stack is a stack of order-1 stacks. Thus we can write \([\text{main}]\) to denote the order-2 stack containing only the order-1 stack \([\text{main}]; [\text{main}]\) is an order-1 stack containing only the character `main`. In general \( \Sigma \) 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.

1. The first step is to rewrite `main` using `main \( \rightarrow M \times nil \)`. Since \((M \times nil)\) is a subterm of our recursion scheme, we have \((M \times nil) \in \Sigma \) and we simply rewrite the stack \([\text{main}]\) to \([M \times nil]\).
2. 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 \times M))`) onto the stack, resulting in the third stack in Figure 2.
3. Note that we do not instantiate the variable \( x \), hence we use only the subterms appearing in the recursion scheme.
4. 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 (\( \cdot \) \( \cdot \)) (\( \cdot \) \( \cdot \))` we have to output the terminal `or` and then (non-deterministically) choose a branch of the tree to follow. Let us choose \((A \times M)\). Hence, the CPDA outputs the terminal `or` and rewrites the top term to \((A \times M)\).
5. Next we make a call to the `A` function, pushing its body on to the stack, and then pick out the `(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 `(error)`. To be able to do this, we have to know the value of `\( \phi \)`. 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 `(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 `\( \phi \)`. While doing this we also create a `collapse link`, pictured as an arrow from `M` to the term `(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 `(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 `(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, 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 \( \Sigma \) 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.
Definition 3.1 (Order-\(\eta\) Collapsible Stacks). Given a finite set of stack characters \(\Sigma\), an order-0 stack is simply a character \(a \in \Sigma\). An order-\(\eta\) stack is a sequence \(w = [w_\eta \ldots w_1]\), such that each \(w_i\) is an order-(\(\eta - 1\)) stack and each character \(a\) on the stack is augmented with a collapse link. The top-most stack is denoted as superscripts. Often (as we have done for the stack character \(a\)) we will omit these superscripts for readability.

Collapse links point to positions in the stack. Before describing them formally, we give an informal description and some basic definitions. An order-\(\eta\) stack can be represented naturally as an edge-labelled word-graph over the alphabet \(\{(i_n - 1), \ldots, [1], \ldots, i_n\}\) \(\cup\) \(\Sigma\), with additional collapse-links pointing from a stack character in \(\Sigma\) to the beginning of the graph representing the target of the link. For technical convenience we do not use \([n]\) or \([0]\) symbols (these appear uniquely at the beginning and end of the stack). An example order-3 stack is given in Figure 1 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-\(\eta\) stack \([w_\eta \ldots w_1]\), we define

\[
\begin{align*}
top_{\eta+1}([w_\eta \ldots w_1]) &= [w_\eta \ldots w_1] \\
top_{\eta}([w_\eta \ldots w_1]) &= w_i & \text{if } \ell > 0 \\
top_{\eta}([w_\eta \ldots w_1]) &= [0] & \text{otherwise} \\
top_{\eta}([w_\eta \ldots w_1]) &= top_\eta(w_i) & \text{if } k < n \text{ and } \ell > 0
\end{align*}
\]

noting that \(top_\eta(w)\) is undefined if \(top_{\eta+1}(w)\) is empty for any \(\ell > k\). We also remove the top portion of a \(top_k\) stack using

\[
\begin{align*}
bot_k([w_\eta \ldots w_1]) &= [w_\eta \ldots w_1], & \text{if } k \leq \ell \leq n \text{ and } i > 0
\end{align*}
\]

Finally the following notation appends a stack on top of another.

Given an order-\(k\) stack \(v = [v_k \ldots v_1]\) and an order-(\(k' - 1\)) stack \(u\) (with \(k' \leq k\)), we define \(u \cdot v = [u v_k \ldots v_1]\) if \(k' = k\) and \(u \cdot v = [u v_k \ldots v_1]\) if \(k' < k\).

The following operations apply to an order-\(k\) collapsible stack.

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

1. \(pop_\eta(u \cdot k; v) = v\).
2. \(push_k(u \cdot k; v) = v \cdot k(u \cdot k; v)\).
3. \(collapse_k(w) = bot_k(w)\) where \(top_1(w) = a^{(k,i)}\) for some \(i\).
4. \(push_k^\eta(w) = b^{(k,\ell - 1)}; v\) where \(top_{\eta+1}(w) = [w_\eta \ldots w_1]\),
5. \(rew_k(a^{(k,i)}; v) = b^{(k,i)}; v\).

Note that, for a \(push_k\) operation, links outside of \(u = top_{\eta+1}(w)\) point to the same destination in both copies of \(u\), while links pointing within \(u\) point within the respective copies of \(u\). Since \(collapse_k\) would always be equivalent to \(pop_\eta\), 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 [11].

3.1.2 Collapsible Pushdown Systems

We are now ready to define collapsible pushdown systems.

Definition 3.3 (Collapsible Pushdown Systems). An order-\(\eta\) collapsible pushdown system \((\eta\text{-CPDS})\) is a tuple \(\mathcal{C} = (P, \Sigma, R)\) where \(P\) is a finite set of control states, \(\Sigma\) is a finite stack alphabet, and \(R \subseteq P \times \Sigma \times \Omega_{\eta} \times P\) is a set of rules.

A configuration of a CPDS is a pair \((p, w)\) where \(p \in P\) and \(w \in Stack_{\eta}.\) We denote by \((p, w) \rightarrow (p', w')\) 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 \((p_0, w_0) \rightarrow \cdots \rightarrow (p_e, w_e)\).
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 $A = (Q_n, \ldots, Q_1, \Sigma, \Delta_n, \ldots, \Delta_1, F_n, \ldots, F_1)$ is a tuple where $\Sigma$ is a finite stack alphabet, and

1. for all $n \geq k \geq 2$, we have $Q_k$ is a finite set of states, $F_k \subseteq Q_k$ is a set of accepting states, and $\Delta_k \subseteq Q_k \times Q_{k-1} \times 2^Q_k$ 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. $Q_1$ is a finite set of states, $F_1 \subseteq Q_1$ a set of accepting states, and $\Delta_1 \subseteq \bigcup_{2 \leq k \leq n} (Q_1 \times \Sigma \times 2^Q_k \times 2^Q_1)$ a transition relation.

The sets $Q_k$ 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 $q \xrightarrow{a} q'$ from $q$ to $Q$ for some $k > 1$ requires that the top $k-1$ stack is accepted from $q' \in Q_{k-1}$ and the rest of the stack is accepted from $Q$. At order-1, a transition $(q, a, Q_{col})$ has the additional requirement that the link stacked to by $a$ is accepted from $Q_{col}$. A stack is accepted if a subset of $F_k$ is reached at the end of each order-$k$ stack. We write $w \in Q_{col}(A)$ 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 3 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 $Q_2 \subseteq F_2$ for the run to be accepting.

The transitions used are $q_1 \xrightarrow{a_{col}} Q_{col}$ for $q_1 \in \Delta_1$. See Section 3.2 for further examples.

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 $q_3 \xrightarrow{a_{col}} (Q_1, q_2, Q_3)$ to capture the transitions shown in Figure 3. 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 $q_3 \xrightarrow{a_{col}} (Q_2, Q_3)$ and $q_3 \xrightarrow{a_{col}} (Q_3).$

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

$q \xrightarrow{a_{col}} (Q_1, \ldots, Q_k) \text{ and } q \xrightarrow{a_{col}} (Q_{k+1}, \ldots, Q_n).$

In the first case, there exist $q_{k-1}, \ldots, q_1$ such that $q \xrightarrow{q_{k-1}} Q_k \in \Delta_k, q_{k-1} \xrightarrow{q_k} Q_{k-1} \in \Delta_{k-1}, \ldots, q_1 \xrightarrow{a_{col}} Q_1 \subseteq \Delta_1.$

In the second case there exist $q_{k-1}, \ldots, q_{k+1}$ with $q \xrightarrow{q_{k-1}} Q_k \in \Delta_k, q_{k-1} \xrightarrow{q_k} Q_{k-1} \in \Delta_{k-1}, \ldots, q_1 \xrightarrow{a_{col}} Q_1 \subseteq \Delta_1.$

3.2.2 Representing Sets of Transitions

Let $\Delta_k^Q$ denote the set of all order-$k$ long-form transitions $q \xrightarrow{a_{col}} (Q_1, \ldots, Q_k)$ of order-$k$. For a set $T = \{t_1, \ldots, t_k\} \subseteq \Delta_k^Q$, we say $T$ is of the form

$q \xrightarrow{a_{col}} (Q_1, \ldots, Q_k)$

whenever $Q = \{q_1, \ldots, q_t\}$ and for all $1 \leq i \leq k$ we have $t_i = q_i \xrightarrow{a_{col}} (Q_1, \ldots, Q_k)$ and $Q_{col} = \bigcup_{1 \leq i \leq k} Q_i^k$ for all $1 \leq k' \leq k$, $Q_{col} = \bigcup_{1 \leq i \leq k} Q_i^k$. Because a link can only be of one order, we insist that $Q_{col} \subseteq Q_{k'}$ for some $1 \leq k' \leq n$.

3.3 Representing Sets of Configurations

We define a notion of $P$-multi-automata for representing sets of configurations of collapsible pushdown systems.

Definition 3.5 (P-Multi Stack Automata). Given an order-$n$ CPDS with control states $P$, a P-multi stack automaton is an order-$n$ stack automaton $A = (Q_n, \ldots, Q_1, \Sigma, \Delta_n, \ldots, \Delta_1, F_n, \ldots, F_1)$ such that for each $p \in P$ there exists a state $q_p \in Q_n$.

A state is initial if it is of the form $q_i \in Q_n$ for some control state $p$ or if it is a state $q_k \in Q_k$ for $k \leq n$ such that there exists a transition $q_{k+1} \xrightarrow{a} q_{k+1} \in \Delta_{k+1}$. The language of a P-multi stack automaton $A$ is the set $L(A) = \{p, w \mid w \in L_{q_p}(A)\}.$

3.4 Basic Saturation Algorithm

Our algorithm computes the set $Pre^n_c(A_0)$ of a collapsible pushdown system $C$ and a 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 $Pre^n_c(A_0)$ be the smallest set with $Pre^n_c(A_0) \supseteq L(A_0)$, and $Pre^n_c(A_0) \supseteq \{(p, w) \mid \exists (p, w) \xrightarrow{a} (p', w) \in Pre^n_c(A_0)\}.$

We begin with $A_0$ and iterate a saturation function $II$ --- 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 $q_n \xrightarrow{a_{col}} (Q_1, \ldots, Q_n)$ to be added to the automaton.

Recall this represents $q \xrightarrow{a_{col}} (Q_1, \ldots, Q_n)$.

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 \( t = q \xrightarrow{a}_{Q_{col}} (Q_1, \ldots, Q_n) \) we will define the justification \( J(t) \) to be either \( 0 \) (indicating the transition is in \( A_0 \)), a pair \((r, i)\), a tuple \((r, i, t)\) 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 \( q_1 \xrightarrow{a}_{Q_{col}} Q_1 \) associated to each \( t \).

The Saturation Function

We are now ready to recall the saturation function \( \Pi \) for a given \( C = (P, \Sigma, R) \). 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 \((p, a, r e w, p')\) which leads to the addition of a transition \( q_0 \xrightarrow{a}_{Q_{col}} (Q_1, \ldots, Q_n) \) whenever there already existed a transition \( q_{p'} \xrightarrow{b}_{Q_{col}} (Q_1, \ldots, Q_n) \). 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 \( q_0 \) with \( a \) on top whenever we had an accepting run from \( q_{p'} \) with \( b \) on top. We give examples of intuition and the more complex steps in Section 6 which may be read alongside the definition below.

Definition 3.6 (The Saturation Function \( \Pi \)). Given an order-\( n \) stack automaton \( A_0 \) we define \( A_{i+1} = \Pi(A_i) \). The state-sets of \( A_{i+1} \) are defined implicitly by the transitions which are in \( A_i \) plus, for each \( r = (p, a, r e w, p') \in R \),

1. when \( o = pop_k \), for each \( q_{p'} \xrightarrow{a}_{Q_{col}} (Q_k, \ldots, Q_n) \) in \( A_i \), add \( t = q_0 \xrightarrow{a}_{(\emptyset, \ldots, \emptyset)} (Q_k, \ldots, Q_n) \) 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 \( t = q_{p'} \xrightarrow{a}_{Q_{col}} (Q_1, \ldots, Q_k, \ldots, Q_n) \) and \( T \) of the form \( Q_k \xrightarrow{a}_{Q_{col}} (Q_1', \ldots, Q_k') \) in \( A_i \), add to \( A_{i+1} \) the transition \( t' = q_0 \xrightarrow{a}_{Q_{col}} (Q_1 \cup Q_1', \ldots, Q_{k-1} \cup Q_{k-1}', \ldots, Q_n) \)

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 \( t = q_0 \xrightarrow{a}_{(\emptyset, \ldots, \emptyset)} (Q_k, \ldots, Q_n) \) to \( A_{i+1} \) if it does not exist, and when \( k < n \), for each transition \( q_{p'} \xrightarrow{a}_{Q_{col}} (Q_k, \ldots, Q_n) \) in \( A_i \), add to \( A_{i+1} \) the transition \( t = q_0 \xrightarrow{a}_{(\emptyset, \ldots, \emptyset)} (Q_k, \ldots, Q_n) \) if it does not already exist. In all cases, if \( t \) is added, set \( J(t) = (r, i + 1) \).
4. when \( o = push_b \), for all transitions \( t = q_{p'} \xrightarrow{a}_{Q_{col}} (Q_1, \ldots, Q_n) \) and \( T = Q_1 \xrightarrow{a}_{Q_{col}} Q_1' \) in \( A_i \) with \( Q_{col} \subseteq Q_k \), add to \( A_{i+1} \) the transition \( t' = q_0 \xrightarrow{a}_{Q_{col}} (Q_1', Q_2, \ldots, Q_k \cup Q_{col}, \ldots, Q_n) \),

and set \( J(t') = (r, t, T, i + 1) \) if \( t' \) is not already in \( A_{i+1} \).
5. when \( o = rew \) for each transition \( t = q_{p'} \xrightarrow{a}_{Q_{col}} (Q_1, \ldots, Q_n) \) in \( A_i \), add to \( A_{i+1} \) the transition \( t' = q_0 \xrightarrow{a}_{Q_{col}} (Q_1, \ldots, Q_n) \), 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 \( n\text{-EXPTIME} \). When \( A_0 \) satisfies a “non-alternating” property (e.g. when we’re only interested in reaching a designated control state), we can restrict \( \Pi \) to only add transitions where \( Q_n \) has at most one element, giving \( (n-1)\text{-EXPTIME} \) complexity. In all cases saturation is linear in the size of \( \Sigma \).

4. Examples of Saturation

As an example of saturation, consider a CPDS with the run \( p = (p_1, [b, c] [d]) \) push \( p_2 \xrightarrow{ab} \) collapse \( p_3 \xrightarrow{c} \) pop pop \( p_5 \xrightarrow{d} \).

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

Initial Automaton

We begin at the top of Figure 5 with a stack automaton containing the transitions \( q_{p_3} \xrightarrow{a} \emptyset \) and \( q_1 \xrightarrow{d} \emptyset \), which we write \( q_0 \xrightarrow{d} \emptyset \). This gives the pictured run over \((p_5, [d]) \).

Rule \((p_2, c, pop_2, p_5)\)

When the saturation step considers such a pop rule, it adds \( q_{p_4} \xrightarrow{a} \emptyset \). We add such a transition because we only require the top order-1 stack (removed by pop2) to have the top character \( c \) (hence \( \emptyset \) is the next order-1 label), and after the pop2 the remaining stack needs to be accepted from \( q_{p_5} \) (hence \( q_{p_4} \) is the next order-2 label). This new transition allows us to construct the next run over \((p_4, [c] [d]) \). Figure 5 shows the labelled \( \{q_{p_4}\} \) in the run comes from the collapse link on the topmost 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 \( q_{p_3} \xrightarrow{a} \emptyset \). We also have \( \emptyset \xrightarrow{\emptyset} \emptyset \) (trivially) accepting the second order-1 stack. Any push2 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 \( q_{p_2} \xrightarrow{a} \emptyset \) which is used to form the shown run over \((p_2, [ab] [ab] [c] [d]) \) (collapse links omitted).

Rule \((p_1, b, push_1^2, p_2)\)

The run from \( q_{p_3} \) in Figure 5 begins with \( q_{p_2} \xrightarrow{a} \emptyset \) and \( \emptyset \xrightarrow{\emptyset} \emptyset \). Note that the \( push_2 \) gives a stack with \( \emptyset \) 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 \( push_2 \) rules creates \( q_{p_1} \xrightarrow{\emptyset} \emptyset \cup \{q_{p_4}\} \). This can be used to construct an accepting run over \((p_1, [b, c] [d]) \).
5. Initial Forward Analysis

In this section we distinguish an error state \( p_{err} \) and we are interested only in whether \( C \) can reach a configuration of the form \( (p_{err}, w) \) (hence our \( A_0 \) is "non-alternating"). This suffices to capture the same safety (reachability) properties of recursion schemes as \( \text{TRecS} \). We fix a stack-automaton \( E \) recognising all error configurations (those with the state \( \text{Error} \)). We write \( \text{Post}_E^2 \) for the set of configurations reachable by \( E \) from the initial configuration. This set cannot be represented precisely by a stack automaton \( 5 \) (for instance using \( \text{push}_{a,b} \), we can create \([a^n]_1[b^n]_2 \) from \([a^n]_1[\lambda]_2 \) for any \( n > 0 \). We summarise our approach then give details in Sections 5.1 and 5.2.

It is generally completely impractical to compute \( \text{Pre}_E^2 \) in full (most non-trivial examples considered in our experiments would time-out). For our saturation algorithm to be usuable in practice, it is therefore essential that the space search is restricted, which we achieve by means of an initial forward analysis of the CPDS. Ideally we would compute only \( \text{Pre}_E^2 \cap \text{Post}_E^2 \). Since this cannot be represented by an automaton, we instead compute a sufficient approximation \( T \) (ideally a strict subset of \( \text{Pre}_E^2 \)) where:

\[
\text{Pre}_E^2 \cap \text{Post}_E^2 \subseteq T \subseteq \text{Pre}_E^2.
\]

The initial configuration will belong to \( T \) iff it can reach a configuration recognised by \( E \). Computing such a \( T \) is much more feasible.

We first compute an over-approximation of \( \text{Post}_E^2 \). For this we use a summary algorithm \( 6 \) (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 \( C' \) 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 \( T = \text{Pre}_{E'}^2 \) (computable by saturation for \( C' \)) since it satisfies the conditions above. This is what we mean 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 \( \text{Post}_E^2 \) about how pops and collapses might be used during a run to \( p_{err} \). Our \( C' \) is then restricted to a model \( C'' \) that 'guards' its rules by these contextual constraints. Taking \( T = \text{Pre}_{E''}^2 \) we have a \( T \) smaller than \( \text{Pre}_E^2 \), but still satisfying our sufficient conditions. In fact, \( C'' \) will be a 'guarded CPDS' (defined in the next subsection). We cannot compute \( \text{Pre}_{E''}^2 \) precisely for a guarded CPDS, but we can adjust saturation to compute \( T \) such that \( \text{Pre}_{E''}^2 \subseteq T \subseteq \text{Pre}_{E''}^2 \). This set will thus also satisfy our sufficient conditions.

5.1 Guarded Destruction

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

\[
o^S(u) := \begin{cases} 
  o(u) & \text{if } o(u) \text{ defined and } \text{top}_1(o(u)) \in S \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

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 \((p, a, \text{push}_{a,b}, p')\) fires it must always result in a stack with \( a \) on top, and \((p, a, \text{push}_{b,c}, p')\) produces a stack with \( b \) on top).

Given a CPDS \( C \), we write \( \text{Tri}v(C) \) for the ordinary CPDS that is the trivialisation of \( C \), obtained by replacing each \( \text{pop}_k \) (resp. \( \text{collapse}_k \)) in the rules of \( C \) with \( \text{pop}_k \) (resp. \( \text{collapse}_k \)).

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 image. This thus improves performance.

1. When \( o = \text{pop}_k \) for each \( q_{err} \xrightarrow{a} (Q_{k+1}, \ldots, Q_n) \) in \( A \) such that there is a transition of the form \( q_k \xrightarrow{a} (\ldots, \ldots) \) in \( A \) such that \( b \in S \), add \( q_p \xrightarrow{a} (\emptyset, \ldots, 0, \{q_k\}, Q_{k+1}, \ldots, Q_n) \) to \( A' \).

2. When \( o = \text{collapse}_k \) for each \( q_{err} \xrightarrow{a} (Q_{k+1}, \ldots, Q_n) \) in \( A \) such that there is a transition of the form \( q_k \xrightarrow{b} (\ldots, \ldots) \) in \( A \) with \( b \in S \), add \( q_p \xrightarrow{a} (\emptyset, \ldots, 0, Q_{k+1}, \ldots, Q_n) \) to \( A' \).

E.g., suppose that an ordinary (non-guarded) 2-CPDS has rules (\( p_1, c, \text{collapse}_{2} \), \( p_2, d, \text{collapse}_{2} \)). The original saturation algorithm would process these rules to add the transitions:

\[
q_{p_1} \xrightarrow{c} (\emptyset, \emptyset) \quad \text{and} \quad q_{p_2} \xrightarrow{d} (\emptyset, \emptyset)
\]

Now suppose that the saturation algorithm has produced two transitions of the form \( q_p \xrightarrow{a} (\ldots, \ldots) \) and \( q_{err} \xrightarrow{b} (\ldots, \ldots) \). If a GCDPS had, for example, the rules (\( p_1, c, \text{collapse}_{2} \), \( p_2, d, \text{collapse}_{2} \)), then these same two transitions would be added by the modified saturation algorithm. On the other hand, the rules (\( p_1, c, \text{collapse}_{2} \), \( p_2, d, \text{collapse}_{2} \)) would only result in the first of the two transitions being added.

**Lemma 5.1.** The revised saturation algorithm applied to \( E \) for a GCDPS \( C \) gives a stack automaton recognising \( T \) such that \( \text{Pre}_{E''}^2 \subseteq T \subseteq \text{Pre}_{\text{Tow}(\text{Cap}(E))} \).

**Remark 5.1.** The algorithm may result in a stack-automaton recognising configurations that do not belong to \( \text{Pre}_{E''}^2 \) (although still in \( \text{Pre}_{\text{Tow}(\text{Cap}(E))} \)). This is because a state \( q_k \) having a transition \( q_k \xrightarrow{a} (\ldots, \ldots) \) may also have another transition
edges over-approximate the transitions that a task uniquely determine the order of any link that is mentioned previously. We refer the reader to the long version for details.

5.2 Approximate Reachability Graphs

Remark 5.2. An approximate reachability graph together with two additional algorithms (applied to a configuration with head at which head the new from a configuration described by ) can be computed using an algorithm from a configuration described by to one described by .

An approximate reachability graph for is a structure ( , ) describing an over-approximation of the reachable configurations of . The set of nodes of the graph includes heads of the CPDS, where a head is a pair , and describes configurations of the form , where . The set contains directed edges ( , , ) labelled by rules of .

Remark 5.2. The CPDS rules occurring in the triples in only if the original also does. However, the approximate reachability graph is described in algorithms 1, 2, 4. The main work done is in the function ProcessHeadWithDescriptor. In particular, this is where approximate reaches are added for the and operations. A fully worked example is given in the long version.

5.3 Extracting the Guarded CPDA

Let be an approximate reachability graph for . Let be the set of heads of error configurations, i.e. . We do a simple backwards reachability computation on the finite graph to compute the set , which is defined to be the smallest set satisfying:

The CPDS rules occurring in the triples in can be used to define a pruned CPDS 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 whose guards are non-trivial. It should be clear that the following set of guarded rules can be computed:

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

Lemma 5.2. The GCPDS defined using satisfies: }
Algorithm 1 The Approximate Summary Algorithm

Require: An \( n \)-CPDS with rules \( R \) and heads \( P \times \Sigma \) and initial configuration \( (p_0, [\cdots [a_0] \cdots]) \)
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\((p_0, a_0), (\mathcal{L}, \cdots, \bot, \bot)\)
return Done, \((H, E, B, U)\) will now be as required

Algorithm 2 AddStackDescriptor\((h, (h_n, \cdots, h_1, h_c))\)

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

if \((h_n, \cdots, h_1, h_c) \in B(h)\) then
return Done (Nothing to do)
Call ProcessHeadWithDescriptor\((h, (h_n, \cdots, h_1, h_c))\)
for \( h' \in H \) such that \((h, (h_n', \cdots, h_{k+1}', h_{k+2} \cdots, h_1, h_c)) \in B(h')\) do
Call AddStackDescriptor\((h', (h_{n+1}', \cdots, h_{k+1}', h_{k+2} \cdots, h_1, h_c))\)
return Done

Algorithm 3 ProcessHeadWithDescriptor\((h, (h_n, \cdots, h_1, h_c))\)

Require: A head \( h := (p, a) \in H \) and a stack descriptor \((h_n, \cdots, h_1, h_c) \in B(h)\)
Ensure: All necessary modifications to the graph are made so that it is consistent with \((h_n, \cdots, h_1, h_c) \in B(h)\). 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 \( o' \) such that \( r = (p, a, o, o') \in R \) do
if \( o = \text{rew} \) then
Add \((p', b) \rightarrow H \) and \((p, a, r, (p', b)) \rightarrow E\)
Call AddStackDescriptor\((p', b), (h_n, \cdots, h_1, h_c))\)
else if \( o = \text{push} b \) then
Add \((p', b) \rightarrow H \) and \((p, a, r, (p', b)) \rightarrow E\)
Call AddStackDescriptor\((p', b), (h_n, \cdots, h_2, (p, a, h_k))\)
else if \( o = \text{push} h \) then
Add \((p', a) \rightarrow H \) and \((p, a, r, (p', a)) \rightarrow E\)
Call AddStackDescriptor\((p', a), (h_n, \cdots, h_{k+1}, (p, a, h_k))\)
else if \( o = \text{pop} \) then
with \( h_c = (p, a) \) where \( a, b \neq \bot \) then
Add \((p', a) \rightarrow H \) and \((p, a, r, (p', a)) \rightarrow E\)
Call AddStackDescriptor\((p', a), (h_n, \cdots, h_{k+1}, (p', a))\)
else if \( o = \text{collapse} \) then
\((p, a) \rightarrow H \) and \((p, a, r, (p', a)) \rightarrow E\)
Call AddStackDescriptor\((p, a), (h_n, \cdots, h_{k+1}, (p', a))\)
return Done

6. Counter Example Generation

In this section, we describe an algorithm that given a CPDS \( C \) and a stack automaton \( A_0 \) such that a configuration \((p, w)\) of \( C \) belongs \( F \) rejects \( (A_0) \), constructs a sequence of rules of \( C \) which when applied from \((p, w)\) 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 4 described in Section 4.1.

To construct a trace from \((p_1, [b] [c] [d]) \) to \((p_2, [d]) \) we first note that, when adding the initial transition of the pictured run from \( q_{01} \), the saturation step marked that the transition was added due to the required higher-order summary edge \((h, (h_n', \cdots, h_{k+1}', h_c)) \), \( h_c \in B(h) \) 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 \((h, (h_n', \cdots, h_{k+1}', h_c)) \in B(h') \) then
return (Nothing to do)
Add \((h, (h_n', \cdots, h_{k+1}', h_c)) \rightarrow U\)
for \((h_n, \cdots, h_{k+1}, h_{k+2} \cdots, h_1, h_c) \in B(h)\) do
AddStackDescriptor\((h', (h_{n+1}', \cdots, h_{k+1}', h_{k+2} \cdots, h_1, h_c))\)
return Done

Algorithm 4 AddStackSummary\((h, (h_n, \cdots, h_{k+1}), h')\)

Require: An approximate higher-order summary edge \((h, (h_n, \cdots, h_{k+1}), h') \)
Ensure: \((h, (h_n, \cdots, h_{k+1}), h') \in U \) and that all necessary stack descriptors are added to the appropriate \( B(h') \) for \( h' \in H \)

Require: An approximate higher-order summary edge \((h, (h_n, \cdots, h_{k+1}), h') \)
Ensure: \((h, (h_n, \cdots, h_{k+1}), h') \in U \) and that all necessary stack descriptors are added to the appropriate \( B(h') \) for \( h' \in H \)

1. For \( k = 1 \), we say \( w' \rightarrow_1 w \) if for some \( i \geq 0 \), \( w \) contains strictly fewer transitions in \( \Delta_1 \) justified at step \( i \) than \( w' \) and that for all \( i > j \) they both contain the same number of transitions in \( \Delta_1 \) justified at step \( j \).
2. For \( k > 1 \), we say \( u = \ [u_i, \cdots, u_j] \) to \( v = [v_i, \cdots, v_j] \) if \( \ell' \leq \ell \) and \( u_i = v_i \) for \( i \in [1, \ell'] \) and either \( u_{\ell'} = v_{\ell'} \) or \( u_{\ell'} \rightarrow_{k-1} v_{\ell'} \), or \( \ell' \geq \ell \) and \( u_i = v_i \) for \( i \in [1, \ell - 1] \) and \( u_{\ell} \rightarrow_{k-1} v_{\ell} \) for all \( i \in [\ell, \ell'] \).

Lemma 6.1. For all \( k \in [1, n] \), the relation \( \rightarrow_k \) is well-founded. Namely there is no infinite sequence \( w_0 \rightarrow_{k-1} w_1 \rightarrow_{k} w_2 \rightarrow_{k} \cdots \).

It is possible to show that by following the justifications, from stack \( w \) to \( w' \), we always have \( w \rightarrow_{n} w' \). 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 [23]. 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 \( \Delta_{\text{new}} \) (implemented as a stack), processed, then moved to a set \( \Delta_{\text{done}} \) 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 \( (p, a, \text{push}_x, p') \), processing \( t = q_{\text{col}} \rightarrow (Q_1, \ldots, Q_{2k}, \ldots, Q_{2k}) \) ‘once and once only’ must somehow include adding a new transition whenever there is a set of transitions of the form \( Q_k \rightarrow Q'_k \) in \( A_t \) 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 \( Q_k \times Q_{k-1} \times \Sigma \times 2^{Q_k} \) where \( Q_0 = \{ \perp \} \) and \( Q_i^+ = Q_i \cup \{ \perp \} \) for \( i \geq 1 \). An order-\( k \) target is a tuple \( (Q_k, Q_k^-, Q_{\text{col}}, Q_k^+) \) where \( Q_k \times Q_{\text{col}} \times Q_k^+ \subseteq 2^{Q_k} \times 2^{Q_k} \times 2^{Q_k} \) if \( k \geq 2 \), and if \( k = 1 \):

\[
(Q_1, Q_1^-, a, Q_{\text{col}}, Q_1^+) = \bigcup_{k' = 2}^{n} \left( 2^{Q_1} \times 2^{Q_1} \times 2^{Q_{k'}} \times 2^{Q_1} \right).
\]

The set \( Q_k^c \) is a countdown containing states in \( Q_k \) still awaiting a transition. We always have \( Q_k^c \subseteq Q_k \) and \( (Q_1 \setminus Q_k^c) \rightarrow Q_1^c \). Likewise, an order-\( 1 \) target \( (Q_1^-, Q_1^+, a, Q_{\text{col}}, Q_1) \) will satisfy \( (Q_1 \setminus Q_1^+) \rightarrow Q_1 \). A target is complete if \( Q_k^c = \emptyset \) or \( Q_k^c = \emptyset \).

A trip-wire of order-\( k \) is an order-\( k \) source-target pair of the form \( (\perp, ..., Q_k, (Q_k, ..., \perp)) \) when \( k \geq 2 \) or \( (\perp, ..., a, Q_k, (Q_k, ..., a, \perp)) \) 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 \( (q_k, a, Q_k) \) (the second component for simplicity of description) and \( (Q_k, \emptyset, Q_{\text{col}}, Q_k^+) \) this means we’ve found a set of transitions witnessing \( Q_k \rightarrow Q_k^+ \) and should now look for transitions from \( Q_{\text{col}} \). Hence the algorithm creates a new source and target for the order-\( (k-1) \) state-set \( Q_{\text{col}} \). 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 6 gives the main loop and introduces the global sets \( \Delta_{\text{new}} \) and \( \Delta_{\text{done}} \), and two arrays \( U_{\text{rc}}(k) \) and \( U_{\text{arr}}(k) \) containing sources and targets for each order. Omitted are loops processing \( \text{pop}_n \) and \( \text{collapse}_n \) 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 \( \Delta_{\text{done}} \) to create further targets. E.g., a new target \( (Q_k, Q_k^c, Q_{\text{col}}, Q_k^+) \) combines with an existing \( q_k \rightarrow Q_k^+ \) to create a new target \( (Q_k, Q_k^c \setminus \{ q_k \}, Q_{\text{col}} \cup \{ q_k \-1 \}, Q_k^c \setminus Q_k^c) \). (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 \( C \) and stack automaton \( A_0 \), let \( A \) be the result of Algorithm 6. We have \( L(A) = \text{Pre}_C(A_0) \).

8. Experimental Results

We compared C-SHORe with the current state-of-the-art verification tools for higher-order recursion schemes (HORS): TReC\( S \) [23], GTRRecS2 [42] (the successor of [24]), and TravMC [38]. Benchmarks are from the TReC\( S \) 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 3, we can construct a reachability query over a CPDS, where the reachability of a control state \( p_{\text{err}} \) 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 TReC\( S \) (version 1.34) and GTRRecS2 (version 3.17) were compiled with the OCaml version 4.00.1 compilers. On negative examples, GTRRecS2 was run with its \(-n\text{msg} \) 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 TReC\( S \). We showed reported times in seconds for TReC\( S \) (T), GTRRecS2 (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 (Cepds), and building the approximation graph (Capprox). Capprox is part of Cepds, and the full time (C) is the sum of Ctran and Cepds.

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-
The saturation technique has proved popular in the literature for HORS model-checking. In addition to various implementations, efficiency implementing PDSolver [15] (for order-1 pushdown systems) indicates that the naive approach is at least an order of magnitude slower than the techniques we generalised in Section 6.

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 [16], parity and μ-calculus conditions [18], and concurrent systems [11], as well as weighted pushdown systems [3]. In addition to various implementations, efficient versions of these algorithms have also been developed [11, 17].

The saturation algorithm for CPDS that we introduced in [7], extending and improving [14], follows a number of papers solving parity games on the configuration graphs of higher-order automata [6, 9, 11, 16]. While only handling reachability, it suggests a more promising capacity to scale on practical incarnations and a number of significant optimisations, such as using a forwards analysis to guide the backwards search.

C-SHORE seems to show a more promising capacity to scale on the fewest examples despite not always terminating in the fastest time. However, 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 we generalised in Section 6.
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-SHOREs.

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 \cite{KKK12} and CFA2 \cite{KK12} 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 \cite{JKM11}. Finally, Hopkins et al. have produced tools for equivalence checking fragments of ML and Idealized Algol up to order-3 \cite{HPR09,HPR11}.

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, Lukasz 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

\[ \text{References} \]


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


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