# Unboundedness and Downward Closures of Higher-Order Pushdown Automata

Matthew Hague
Royal Holloway, University of London, UK
matthew.hague@rhul.ac.uk

Jonathan Kochems
Department of Computer Science, University of Oxford, UK
jonathan.kochems@googlemail.com

C.-H. Luke Ong
Department of Computer Science, University of Oxford, UK
Luke.Ong@cs.ox.ac.uk

## Abstract

We show the diagonal problem for higher-order pushdown automata (HOPDA), and hence the simultaneous unboundedness problem, is decidable. From recent work by Zetzsche this means that we can construct the downward closure of the set of words accepted by a given HOPDA. This also means we can construct the downward closure of the Parikh image of a HOPDA. Both of these consequences play an important rôle in verifying concurrent higher-order programs expressed as HOPDA or safe higher-order recursion schemes.

***Categories and Subject Descriptors*** F.4.3 [*Theory of Computation*]: Mathematic Logic and Formal Languages

***Keywords*** Higher-Order Programs, Functional Programs, Model-Checking, Verification, Formal Languages, Downward Closures, Parikh Images, Concurrency, Automata, Pushdown Automata

## Acknowledgments

We thank Georg Zetzsche for keeping us up to date with his work, Jason Crampton for knowing about logarithms when they were most required, and Chris Broadbent for discussions. This work was supported by the Engineering and Physical Sciences Research Council [EP/K009907/1 and EP/M023974/1].

## 1. Introduction

Recent work by Zetzsche [41] has given a new technique for computing the downward closure of classes of languages. The downward closure $\downarrow(\mathcal{L})$ of a language $\mathcal{L}$ is the set of all subwords of words in $\mathcal{L}$ (e.g. $aa$ is a subword of $babab$). It is well known that the downward closure is regular for any language [20]. However, there are only a few classes of languages for which it is known how to compute this closure. In general it is not possible to compute the downward closure since it would easily lead to a solution to the halting problem for Turing machines.

However, once a regular representation of the downward closure has been obtained, it can be used in all kinds of analysis, since regular languages are well behaved under all kinds of transformations. For example, consider a system that waits for messages from a complex environment. This complex environment can be abstracted by the downward closure of the messages it sends or processes it spawns. This corresponds to a lossy system where some messages may be ignored (or go missing), or some processes may simply not contribute to the remainder of the execution. In many settings – e.g. the analysis of safety properties of certain kinds of systems – unread messages or unscheduled processes do not effect the precision of the analysis. Since many types of system permit synchronisation with a regular language, this environment abstraction can often be built into the system being analysed.

Many popular languages such as JavaScript, Python, Ruby, and even C++, include higher-order features – which are increasingly important given the popularity of event-based programs and asynchronous programs based on a continuation or callback style of programming. Hence, the modelling of higher-order function calls is becoming key to analysing modern day programs.

A popular approach to verifying higher-order programs is that of *recursion schemes* and several tools and practical techniques have been developed [5, 8, 24, 25, 27, 31, 35, 39]. Recursion schemes have an automaton model in the form of collapsible pushdown automata (CPDA) [18] which generalises an order-2 model called 2-PDA with links [1] or, equivalently, panic automata [23]. When these recursion schemes satisfy a syntactical condition called *safety*, a restriction of CPDA called *higher-order pushdown automata (HOPDA or $n$-PDA for order-$n$ HOPDA)* is sufficient [22, 30]. HOPDA can be considered an extension of pushdown automata to a "stack of stacks" structure. It remains open as to whether CPDA are strictly more expressive than nondeterministic HOPDA when generating languages of words. It is known that, at order 2, nondeterministic HOPDA and CPDA generate the same word languages [1]. However, there exists a language generated by a deterministic order-2 CPDA that cannot be generated by a deterministic HOPDA of any order [32].

It is well known that concurrency and first-order recursion very quickly leads to undecidability (e.g. [34]). Hence, much recent research has focussed on decidable abstractions and restrictions (e.g. [4, 10, 12, 13, 16, 21, 28, 29, 37]). Recently, these results have been extended to concurrent versions of CPDA and recursion schemes (e.g. [15, 26, 33, 36]). Many approaches rely on combining representations of the Parikh image of individual automata (e.g. [12, 16, 17]). However, combining Parikh images of HOPDA quickly leads to undecidability (e.g. [17]). In many cases, the downward closure of the Parikh image is an adequate abstraction.

Computing downward closures appears to be a hard problem. Recently Zetzsche introduced a new general technique for classes of automata effectively closed under rational transductions – also referred to as a *full trio*. For these automata the downward closure is computable iff the *simultaneous unboundedness problem (SUP)* is decidable.

DEFINITION 1.1 (SUP [41]). *Given a language $\mathcal{L} \subseteq a_1^* \dots a_\alpha^*$ does $\downarrow(\mathcal{L}) = a_1^* \dots a_\alpha^*$?*

THEOREM 1.1. *[41, Theorem 1] Let $\mathcal{C}$ be class of languages that is a full trio. Then downward closures are computable for $\mathcal{C}$ if and only if the SUP is decidable for $\mathcal{C}$.*

Zetzsche used this result to obtain the downward closure of languages definable by 2-PDA, or equivalently, languages definable by *indexed grammars* [2]. Moreover, for classes of languages closed under rational transductions, Zetzsche shows that the simultaneous unboundedness problem is decidable iff the *diagonal problem* is decidable. The diagonal problem was introduced by Czerwiński and Martens [11]. Intuitively, it is a relaxation of the SUP that is insensitive to the order the characters are output. For a word $w$, let $|w|_a$ be the number of occurrences of $a$ in $w$.

DEFINITION 1.2 (Diagonal Problem [11]). *Given language $\mathcal{L}$ we define*

$$\textit{Diagonal}_{a_1,\dots,a_\alpha}(\mathcal{L}) = \forall m.\exists w \in \mathcal{L}.\forall 1 \leq i \leq \alpha.|w|_{a_i} \geq m \ .$$

*The diagonal problem asks if $\textit{Diagonal}_{a_1,\dots,a_\alpha}(\mathcal{L})$ holds of $\mathcal{L}$.*

COROLLARY 1.1 (Diagonal Problem and Downward Closures). *Let $\mathcal{C}$ be class of languages that is a full trio. Then downward closures are computable for $\mathcal{C}$ if and only if the diagonal problem is decidable for $\mathcal{C}$.*

*Proof.* The only-if direction follows from Theorem 1.1 since given a language $\mathcal{L} \subseteq a_1^* \dots a_\alpha^*$ the diagonal problem is immediately equivalent to the SUP. In the if direction, the result follows since $\mathcal{L}$ satisfies the diagonal problem iff $\downarrow(\mathcal{L})$ also satisfies the diagonal problem. Since the diagonal problem is decidable for regular languages and $\downarrow(\mathcal{L})$ is regular, we have the result. □

In this work, we generalise Zetzsche's result for 2-PDA to the general case of $n$-PDA. We show that the diagonal problem is decidable. Since HOPDA are closed under rational transductions, we obtain decidability of the simultaneous unboundedness problem, and hence a method for constructing the downward closure of a language defined by a HOPDA.

COROLLARY 1.2 (Downward Closures). *Let $P$ be an $n$-PDA. The downward closure $\downarrow(\mathcal{L}(P))$ is computable.*

*Proof.* From Theorem 6.3 (proved in the sequel), we know that the diagonal problem for HOPDA is decidable. Thus, using Corollary 1.1, we can construct the downward closure of $P$. □

This result provides an abstraction upon which new results may be based. It also has several immediate consequences:

1. decidability of separability by piecewise testable languages, from Czerwiński and Martens [11],
2. decidability of reachability for parameterised concurrent systems of HOPDA communicating asynchronously via a shared global register, from La Torre *et al.* [38],
3. decidability of finiteness of a language defined by a HOPDA, and
4. computability of the downward closure of the Parikh image of a HOPDA.

We present our decidability proof in two stages. First we show how to decide $\text{Diagonal}_a(P)$ for a single character and HOPDA $P$ in Sections 3 and 4. In Sections 5, 6, and 7 we generalise our techniques to the full diagonal problem.

In Section 3.1 we give an outline of the proof techniques for deciding $\text{Diagonal}_a(P)$. In short, the outermost stacks of an $n$-PDA are created and destroyed using $\text{push}_n$ and $\text{pop}_n$ operations. These $\text{push}_n$ and $\text{pop}_n$ operations along a run of an $n$-PDA are "well-bracketed" (each $\text{push}_n$ has a matching $\text{pop}_n$ and these matchings don't overlap). The essence of the idea is to take a standard tree decomposition of these well-bracketed runs and observe that each branch of such a tree can be executed by an $(n-1)$-PDA. We augment this $(n-1)$-PDA with "regular tests" that allow it to know if, each time a branch is chosen, the alternative branch could have output some $a$ characters. If this is true, then the $(n-1)$-PDA outputs a single $a$ to account for these missed characters. We prove that, although the $(n-1)$-PDA outputs far fewer characters, it can still output an unbounded number iff the $n$-PDA could. Hence, by repeating this reduction, we obtain a 1-PDA, for which the diagonal problem is decidable since it is known how to compute their downward closures [9, 40].

In Section 6.1 we outline the generalisation of the proof to the full problem $\text{Diagonal}_{a_1,\dots,a_\alpha}(P)$. The key difficulty is that it is no longer enough for the $(n-1)$-PDA to follow only a single branch of the tree decomposition: it may need up to one branch for each of the $a_1, \dots, a_\alpha$. Hence, we define HOPDA that can output trees with a bounded number ($\alpha$) of branches. We then show that our reduction can generalise to HOPDA outputting trees (relying essentially on the fact that the number of branches is bounded).

## 2. Preliminaries

### 2.1 Downward Closures

Given two words $w = \gamma_1 \dots \gamma_m \in \Sigma^*$ and $w' = \sigma_1 \dots \sigma_l \in \Sigma^*$ for some alphabet $\Sigma$, we write $w \leq w'$ iff there exist $i_1 < \dots < i_m$ such that for all $1 \leq j \leq m$ we have $\gamma_j = \sigma_{i_j}$. Given a set of words $\mathcal{L} \subseteq \Sigma^*$, we denote its downward closure $\downarrow(\mathcal{L}) = \{w \mid w \leq w' \in \mathcal{L}\}$.

### 2.2 Trees

A $\Sigma$-labelled finite tree is a tuple $T = (D, \lambda)$ where $\Sigma$ is a set of node labels, and $D \subset \mathbb{N}^*$ is a finite set of nodes that is prefix-closed, that is, $\eta\,\delta \in D$ implies $\eta \in D$, and $\lambda : D \to \Sigma$ is a function labelling the nodes of the tree.

We write $\varepsilon$ to denote the root of a tree (the empty sequence). We also write

$$a[T_1, \dots, T_m]$$

to denote the tree whose root node is labelled $a$ and has children $T_1, \dots, T_m$. That is, we define $a[T_1, \dots, T_m] = (D', \lambda')$ when for each $\delta$ we have $T_\delta = (D_\delta, \lambda_\delta)$ and $D' = \{\delta\eta \mid \eta \in D_\delta\} \cup \{\varepsilon\}$ and

$$\lambda'(\eta) = \begin{cases} a & \eta = \varepsilon \\ \lambda_\delta(\eta') & \eta = \delta\eta' \end{cases} .$$

Also, let $T[a]$ denote the tree $(\{\varepsilon\}, \lambda)$ where $\lambda(\varepsilon) = a$. A *branch* in $T = (D, \lambda)$ is a sequence of nodes of $T$, $\eta_1 \cdots \eta_n$, such that $\eta_1 = \epsilon$, $\eta_n = \delta_1\,\delta_2 \cdots \delta_{n-1}$ is maximal in $D$, and $\eta_{j+1} = \eta_j\,\delta_j$ for each $1 \leq j \leq n-1$.

### 2.3 HOPDA

HOPDA are a generalisation of pushdown systems to a stack-of-stacks structure. An order-$n$ stack is a stack of order-$(n-1)$ stacks. An order-$n$ push operation pushes a new order-$(n-1)$ stack onto the stack that is a copy of the existing topmost order-$(n-1)$ stack. Rewrite operations update the character that is at the top of the topmost stacks.

DEFINITION 2.1 (Order-$n$ Stacks). *The set of order-n stacks $\mathcal{S}^{\Gamma}_n$ over a given stack alphabet $\Gamma$ is defined inductively as follows.*

$$\begin{array}{rcl} \mathcal{S}^{\Gamma}_0 & = & \Gamma \\ \mathcal{S}^{\Gamma}_{k+1} & = & \left\{ [s_1 \dots s_m]_{k+1} \mid \forall i. s_i \in \mathcal{S}^{\Gamma}_k \right\} . \end{array}$$

Stacks are written with the top part of the stack to the left. We define several operations.

$$\begin{array}{rcll} \text{top}_k\big([s_1 \dots s_m]_k\big) & = & s_1 & \\ \text{top}_k\big([s_1 \dots s_m]_n\big) & = & \text{top}_k(s_1) & n > k \\ & & & \\ \text{rew}_\gamma\big([\gamma_1 \dots \gamma_m]_1\big) & = & [\gamma\ \gamma_2 \dots \gamma_m]_1 & \\ \text{rew}_\gamma\ \big([s_1 \dots s_m]_n\big) & = & [\text{rew}_\gamma(s_1)\ s_2 \dots s_m]_n & n > 1 \\ & & & \\ \text{push}_k\big([s_1 \dots s_m]_k\big) & = & [s_1\ s_1 \dots s_m]_k & \\ \text{push}_k\big([s_1 \dots s_m]_n\big) & = & [\text{push}_k(s_1)\ s_2, \dots, s_m]_n & n > k \\ & & & \\ \text{pop}_k\big([s_1 \dots s_m]_k\big) & = & [s_2 \dots s_m]_k & \\ \text{pop}_k\big([s_1 \dots s_m]_n\big) & = & [\text{pop}_k(\ s_1)\ s_2, \dots, s_m]_n & n > k \end{array}$$

and set

$$\text{Ops}_n = \{\text{rew}_\gamma \mid \gamma \in \Gamma\} \cup \{\text{push}_k, \text{pop}_k \mid 1 \leq k \leq n\}$$

to be the set of order-$n$ stack operations.

For example

$$\begin{array}{rcl} \text{push}_2\big(\big[[\gamma\ \sigma]_1\big]_2\big) & = & \big[[\gamma\ \sigma]_1\ [\gamma\ \sigma]_1\big]_2 \\ \text{rew}_\sigma\big(\big[[\gamma\ \sigma]_1\ [\gamma\ \sigma]_1\big]_2\big) & = & \big[[\sigma\ \sigma]_1\ [\gamma\ \sigma]_1\big]_2 . \end{array}$$

DEFINITION 2.2 (HOPDA or $n$-PDA). *An order-n* higher order pushdown automaton (HOPDA or $n$-PDA) *is given by a tuple $(\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \mathcal{F}, p_{\text{in}}, \gamma_{\text{in}})$ where $\mathcal{P}$ is a finite set of control states, $\Sigma$ is a finite output alphabet (that contains the empty word character $\epsilon$), $\Gamma$ is a finite stack alphabet, $\mathcal{R} \subseteq \mathcal{P} \times \Gamma \times \Sigma \times Ops_n \times \mathcal{P}$ is a set of transition rules, $\mathcal{F}$ is a set of accepting control states, $p_{\text{in}} \in \mathcal{P}$ is the initial control state, and $\gamma_{\text{in}} \in \Gamma$ is the initial stack character.*

We write $(p, \gamma) \xrightarrow{a} (p', o)$ for a rule $(p, \gamma, a, o, p') \in \mathcal{R}$.

A configuration of an $n$-PDA is a tuple $\langle p, s \rangle$ where $p \in \mathcal{P}$ and $s$ is an order-$n$ stack over $\Gamma$. We have a transition $\langle p, s \rangle \xrightarrow{a} \langle p', s' \rangle$ whenever we have $(p, \gamma) \xrightarrow{a} (p', o)$, $\text{top}_1(s) = \gamma$, and $s' = o(s)$.

A run over a word $w \in \Sigma^*$ is a sequence of configurations $c_0 \xrightarrow{a_1} \cdots \xrightarrow{a_m} c_m$ such that the word $a_1 \dots a_m$ is $w$. It is an accepting run if $c_0 = \langle p_{\text{in}}, [\![\gamma_{\text{in}}]\!]_n \rangle$ — where we write $[\![\gamma]\!]_n$ for $\left[\cdots [\gamma]_1 \cdots\right]_n$ — and where $c_m = \langle p, s \rangle$ with $p \in \mathcal{F}$. Furthermore, for a set of configurations $C$, we define

$$\text{Pre}^*_P(C)$$

to be the set of configurations $c$ such that there is a run over some word from $c$ to $c' \in C$. When $C$ is defined as the language of some automaton $A$ accepting configurations, we abuse notation and write $\text{Pre}^*_P(A)$ instead of $\text{Pre}^*_P(\mathcal{L}(A))$.

For convenience, we sometimes allow a set of characters to be output instead of only one. This is to be interpreted as outputing each of the characters in the set once (in some arbitrary order). We also allow sequences of operations $o_1; \dots; o_m$ in the rules instead of single operations. When using sequences we allow a test operation $\gamma?$ that only allows the sequence to proceed if the $\text{top}_1$ character of the stack is $\gamma$. All of these extensions can be encoded by introducing intermediate control states.

#### 2.3.1 Regular Sets of Stacks

We will need to represent sets of stacks. To do this we will use automata to recognise stacks. We define the stack automaton model of Broadbent *et al.* [6] restricted to HOPDA rather than CPDA. We will sometimes call these *bottom-up stack automata* or simply *automata*. The automata operate over stacks interpreted as words, hence the opening and closing braces of the stacks appear as part of the input. We annotate these braces with the order of the stack the braces belong to. Let $\Gamma_{[]} = \{[_{n-1}, \dots, [_1, ]_1, \dots, ]_{n-1}\} \uplus \Gamma$. Note, we don't include $[_n, ]_n$ since these appear exclusively at the start and end of the stack.

DEFINITION 2.3 (Bottom-up Stack Automata). *A tuple $A$ is a* bottom-up stack automaton *when $A$ is $(\mathcal{Q}, \Gamma, q_{\text{in}}, \mathcal{Q}_F, \Delta)$ where $\mathcal{Q}$ is a finite set of states, $\Gamma$ is a finite input alphabet, $q_{\text{in}} \in \mathcal{Q}$ is the initial state and $\Delta : (\mathcal{Q} \times \Gamma) \to \mathcal{Q}$ is a deterministic transition function.*

Representing higher order stacks as a linear word graph, where the start of an order-$k$ stack is an edge labelled $[_k$ and the end of an order-$k$ stack is an edge labelled $]_k$, a run of a bottom-up stack automaton is a labelling of the nodes of the graph with states in $\mathcal{Q}$ such that

1. the rightmost (final) node is labelled by $q_{\text{in}}$, and
2. whenever we have for any $\gamma \in \Gamma_{[]}$, and pair of labelled nodes with an edge $q \xrightarrow{\gamma} q'$ then $q = \Delta(q', \gamma)$.

The run is accepting if the leftmost (initial) node is labelled by $q \in \mathcal{Q}_F$. An example run over the word graph representation of $\left[\left[[\gamma\ \sigma]_1\ [\sigma]_1\right]_2\ \left[[\sigma]_1\right]_2\right]_3$ is given in Figure 1.

Let $\mathcal{L}(A)$ be the set of stacks with accepting runs of $A$. Sometimes, for convenience, if we have a configuration $c = \langle p, s \rangle$ of a HOPDA, we will write $c \in \mathcal{L}(A)$ when $s \in \mathcal{L}(A)$.

## 3. The Single Character Case

We assume $\Sigma = \{a, \varepsilon\}$ and use $b$ to range over $\Sigma$. This can be obtained by simply replacing all other characters with $\varepsilon$. We also assume that all rules of the form $(p, \gamma) \xrightarrow{b} (p', o)$ with $o = \text{push}_n$ or $o = \text{pop}_n$ have $b = \varepsilon$. We can enforce this using intermediate control states to first apply $o$ in one step, and then in another output $b$ (the stack operation on the second step will be $\text{rew}_\gamma$ where $\gamma$ is the current top character). We start with an outline of the proof, and then explain each step in detail.

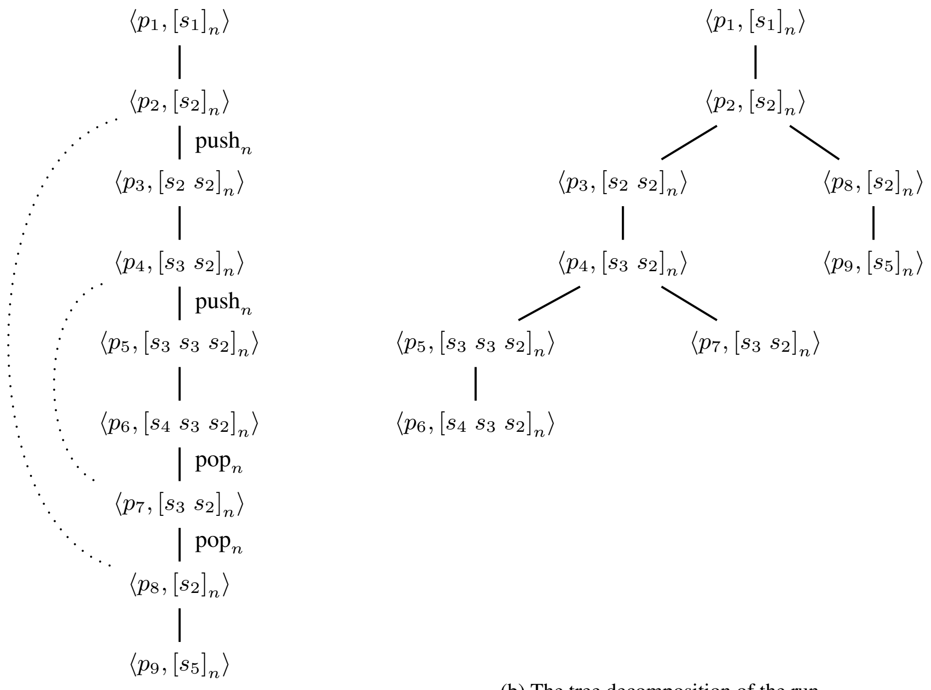
For convenience, we assume acceptance is by reaching a unique control state in $\mathcal{F}$ with an empty stack (i.e. the lowermost stack was removed with a $\text{pop}_n$ and $\mathcal{F} = \{p_f\}$). This can easily be obtained by adding a rule to a new accepting state whenever we have a rule leading to a control state in $\mathcal{F}$. From this new state we can loop and perform $\text{pop}_n$ operations until the stack is empty.

### 3.1 Outline of Proof

The approach is to take an $n$-PDA $P$ and produce an $(n-1)$-PDA $P_{-1}$ that satisfies the diagonal problem iff $P$ does. The idea behind this reduction is that an (accepting) run of $P$ can be decomposed into a tree with out-degree at most 2: each $\text{push}_n$ has a matching $\text{pop}_n$ that brings the stack back to be the same as it was before the $\text{push}_n$; we cut the run at the $\text{pop}_n$ and hang the tail next to the $\text{push}_n$ and repeat this to form a tree from a run. This is illustrated in Figure 2 where nodes are labelled by their configurations, and the $\text{push}_n$ and $\text{pop}_n$ points are marked. The dotted arcs connect nodes matched by their pushes and pops – these nodes have the same stacks. Notice that at each branching point, the left and right subtrees start with the same order-$(n-1)$ stacks on top. Notice also that for each branch, none of its transitions remove the topmost order-$(n-1)$ stack. Hence, we can produce an $(n-1)$-PDA that picks a branch of this tree decomposition to execute and only needs to keep track of the topmost order-$(n-1)$ stack of the $n$-PDA. When picking a branch to execute, the $(n-1)$-PDA outputs a single $a$ if the branch not chosen could have output some $a$ characters. We prove that this is enough to maintain unboundedness.

$$\begin{tikzcd}
q_f \arrow[r, dash, "{[}_2"] & q_{13} \arrow[r, dash, "{[}_1"] & q_{12} \arrow[r, dash, "\gamma"] & q_{11} \arrow[r, dash, "\sigma"] & q_{10} \arrow[r, dash, "{]}_1"] & q_9 \arrow[r, dash, "{[}_1"] & q_8 \arrow[r, dash, "\sigma"] & q_7 \arrow[r, dash, "{]}_1"] & q_6 \arrow[r, dash, "{]}_2"] & q_5 \arrow[r, dash, "{[}_2"] & q_4 \arrow[r, dash, "{[}_1"] & q_3 \arrow[r, dash, "\sigma"] & q_2 \arrow[r, dash, "{]}_1"] & q_1 \arrow[r, dash, "{]}_2"] & q_{\text{in}}
\end{tikzcd}$$

Figure 1: A run over $\left[\left[[\gamma\ \sigma]_1\ [\sigma]_1\right]_2\ \left[[\sigma]_1\right]_2\right]_3$



$\langle p_1, [s_1]_n \rangle$
$\langle p_2, [s_2]_n \rangle$
push$_n$
$\langle p_3, [s_2\ s_2]_n \rangle$
$\langle p_4, [s_3\ s_2]_n \rangle$
push$_n$
$\langle p_5, [s_3\ s_3\ s_2]_n \rangle$
$\langle p_6, [s_4\ s_3\ s_2]_n \rangle$
pop$_n$
$\langle p_7, [s_3\ s_2]_n \rangle$
pop$_n$
$\langle p_8, [s_2]_n \rangle$
$\langle p_9, [s_5]_n \rangle$

(a) a run of $P$ with push$_n$s and pop$_n$s marked.

(b) The tree decomposition of the run

Figure 2: Tree decompositions of runs.

In more detail, we perform the following steps.

1. Instrument $P$ to record whether an $a$ character has been output. Then, using known reachability results, obtain regular sets of configurations from which the current top$_n$ stack can be popped, and moreover, we can know whether an $a$ is output on the way. These tests can be seen as a generalisation of pushdown systems with regular tests introduced by Esparza *et al.* [14].

2. From an $n$-PDA $P$, we define an $(n-1)$-PDA with tests $P_{-1}$ and then an $(n-1)$-PDA $P'$ such that

$$\text{Diagonal}_a(P) \iff \text{Diagonal}_a(P') \ .$$

   The tests will be used to check the branches of the tree decomposition not explored by $P_{-1}$.

3. By repeated applications of the above reduction, we obtain an 1-PDA $P$ for which Diagonal$_a(P)$ is decidable since the downward closure of a context-free grammar (equivalent to 1-PDA) is computable [9, 40] and this is equivalent to the diagonal problem.

The $(n-1)$-PDA with tests $P_{-1}$ will simulate the $n$-PDA $P$ in the following way.

- All operations except for push$_n$ and pop$_n$ will be simulated directly.
- In lieu of performing a push$_n$, $P_{-1}$ will choose to simulate the run of $P$ between the push and its corresponding pop$_n$, or the run of $P$ after the corresponding pop$_n$ has taken place.
  - Tests will be used to determine which control state could appear after the corresponding pop$_n$.
  - If the part of the run not being simulated output some $a$s, then $P$ will output a single $a$ in place of the omitted $a$s.

Although $P_{-1}$ will output far fewer $a$ characters than $P$ (since it does not execute the full run), we show that it still outputs enough $a$s for the language to remain unbounded.

We thus have the following theorem.

THEOREM 3.1 (Decidability of the Diagonal Problem). *Given an $n$-PDA $P$ and output character $a$, whether Diagonal$_a(P)$ holds is decidable.*

*Proof.* We construct via Lemma 3.2 an $(n-1)$-PDA $P'$ such that Diagonal$_a(P)$ iff Diagonal$_a(P')$. We repeat this step until we have a 1-PDA. It is known that Diagonal$_a(P)$ for an 1-PDA is decidable since it is possible to compute the downward closure [9, 40]. □

### 3.2 HOPDA with Tests

When executing a branch of the tree decomposition, to be able to ensure the branch is correct and whether we should output an extra $a$ we need to know how the system could have behaved on the skipped branch. To do this we add tests to the HOPDA that allow it to know if the current stack belongs to a given regular set. We show in the following sections that the properties required for our reduction can be represented as regular sets of stacks. Although we take Broadbent *et al.*'s logical reflection as the basis of our proof, HOPDA with tests can be seen as a generalisation of pushdown systems with regular valuations due to Esparza *et al.* [14].

DEFINITION 3.1 ($n$-PDA with Tests). *Given a sequence of automata $A_1, \ldots, A_m$ recognising regular sets of stacks, an $n$-*PDA with tests *is a tuple $P = (\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \mathcal{F}, p_{\text{in}}, \gamma_{\text{in}})$ where $\mathcal{P}, \Sigma, \Gamma, \mathcal{F}, p_{\text{in}}$, and $\gamma_{\text{in}}$ are as in HOPDA, and*

$$\mathcal{R} \subseteq \mathcal{P} \times \Gamma \times \{A_1, \ldots, A_m\} \times \Sigma \times Ops_n \times \mathcal{P}$$

*is a set of transition rules.*

We write $(p, \gamma, A_i) \xrightarrow{b} (p', o)$ for $(p, \gamma, A_i, b, o, p') \in \mathcal{R}$. We have a transition $\langle p, s\rangle \xrightarrow{b} \langle p', s'\rangle$ whenever $(p, \gamma, A_i) \xrightarrow{b} (p', o) \in \mathcal{R}$ and $\text{top}_1(s) = \gamma$, $s \in \mathcal{L}(A_i)$, and $s' = o(s)$.

We know from Broadbent *et al.* that these tests do not add any extra power to HOPDA. Intuitively, we can embed runs of the automata into the stack during runs of the HOPDA.

THEOREM 3.2 (Removing Tests). *[6, Theorem 3 (adapted)] For every $n$-PDA with tests $P$, we can compute an $n$-PDA $P'$ with $\mathcal{L}(P) = \mathcal{L}(P')$.*

*Proof.* This is a straightforward adaptation of Broadbent *et al.* [6]. A more general theorem is proved in Theorem 6.1. □

### 3.2.1 Marking Outputs

When the HOPDA is in a configuration of the form $\langle p, [s]_n\rangle$ – i.e. the outermost stack contains only a single order-$(n-1)$ stack – we require the HOPDA to be able to know whether,

- for a given $p_1$ and $p_2$, there is a run from $\langle p_1, [s]_n\rangle$ to $\langle p_2, []_n\rangle$ (that is, the HOPDA empties the stack), and
- whether, during the run, an $a$ is output.

Given $P$, we first augment $P$ to record whether an $a$ has been produced. This can be done simply by recording in the control state whether $a$ has been output.

DEFINITION 3.2 ($P_a$). *Given $P = (\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \mathcal{F}, p_{\text{in}}, \gamma_{\text{in}})$ we define*

$$P_a = (\mathcal{P} \cup \mathcal{P}_a, \Sigma, \Gamma, \mathcal{R} \cup \mathcal{R}_a, \mathcal{F} \cup \mathcal{F}_a, p_{\text{in}}, \gamma_{\text{in}})$$

*where*

$$\begin{array}{rcl} \mathcal{P}_a & = & \{p_a \mid p \in \mathcal{P}\} \\ \mathcal{R}_a & = & \left\{(p_a, \gamma) \xrightarrow{b} (p'_a, o) \;\middle|\; (p, \gamma) \xrightarrow{b} (p', o) \in \mathcal{R}\right\} \cup \\ & & \left\{(p, \gamma) \xrightarrow{a} (p'_a, o) \;\middle|\; (p, \gamma) \xrightarrow{a} (p', o) \in \mathcal{R}\right\} \\ \mathcal{F}_a & = & \{p_a \mid p \in \mathcal{F}\} \end{array}$$

It is easy to see that $P$ and $P_a$ accept the same languages, and that $P_a$ is only in a control state $p_a$ if an $a$ has been output.

### 3.2.2 Building the Automata

Fix some $P = (\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \mathcal{F})$ and $P_a = (\mathcal{P}_a, \Sigma, \Gamma, \mathcal{R}_a, \mathcal{F}_a)$. To obtain a HOPDA with tests, we need, for each $p_1, p_2 \in \mathcal{P}$ the following automata. Note, we define these automata to accept order-$(n-1)$ stacks since they will be used in an $(n-1)$-PDA with tests.

1. $A_{p_1,p_2}$ accepting all stacks $s$ such that there is a run of $P$ from $\langle p_1, [s]_n\rangle$ to $\langle p_2, []_n\rangle$,
2. $A^a_{p_1,p_2}$ accepting all stacks $s$ such that there is a run of $P$ from $\langle p_1, [s]_n\rangle$ to $\langle p_2, []_n\rangle$ that outputs at least one $a$.

To do this we will use a reachability result due to Broadbent *et al.* that appeared in ICALP 2012 [7]. This result uses an automata representation of sets of configurations. However, these automata are slightly different in that they read full configurations "top down", whereas the automata of Theorem 3.2 (Removing Tests) read only stacks "bottom up".

It is known that these two representations are effectively equivalent, and that both form an effective boolean algebra [6, 7]. In particular, for a top-down automaton $A$ and a control state $p$ we can build a bottom-up stack automaton $B$ such that $\langle p, s\rangle \in \mathcal{L}(A)$ iff $s \in \mathcal{L}(B)$ and vice versa. We recall the reachability result.

THEOREM 3.3. *[7, Theorem 1 (specialised)] Given an HOPDA $P$ and a top-down automaton $A$, we can construct an automaton $A'$ accepting $Pre^*_P(A)$.*

Let $A_{p,\gamma}$ be a top-down automaton accepting configurations of the form $\langle p, [s]_n\rangle$ where $\text{top}_1(s) = \gamma$. Next, let

$$A_p = \bigcup_{(p',\gamma) \xrightarrow{\varepsilon} (p, \text{pop}_n) \in \mathcal{R}} A_{p',\gamma}$$

and

$$A^a_p = \bigcup_{(p',\gamma) \xrightarrow{\varepsilon} (p, \text{pop}_n) \in \mathcal{R}} A_{p'_a,\gamma}$$

I.e. $A_p$ and $A^a_p$ accept configurations of $P_a$ from which it is possible to perform a $\text{pop}_n$ operation to $p$ and reach the empty stack.

DEFINITION 3.3 ($A_{p_1,p_2}$ and $A^a_{p_1,p_2}$). *Using the preceding notation, given $p_1$ and $p_2$ we define bottom-up automata*

- $A_{p_1,p_2}$ *where* $\mathcal{L}(A_{p_1,p_2}) = \left\{s \;\middle|\; \langle p_1, [s]_n\rangle \in Pre^*_P(A_{p_2})\right\}$ .
- $A^a_{p_1,p_2}$ *where* $\mathcal{L}\left(A^a_{p_1,p_2}\right) = \left\{s \;\middle|\; \langle p_1, [s]_n\rangle \in Pre^*_{P_a}\left(A^a_{p_2}\right)\right\}$ .

It is easy to see both $A_{p_1,p_2}$ and $A^a_{p_1,p_2}$ are regular and representable by bottom-up automata since both

$$\text{Pre}^*_P(A_{p_2}) \quad \text{and} \quad \text{Pre}^*_{P_a}\left(A^a_{p_2}\right)$$

are regular from Theorem 3.3, and bottom-up and top-down automata are effectively equivalent. To enforce only stacks of the form $[s]_n$ we intersect with an automaton $A_1$ accepting all stacks containing a single order-$(n-1)$ stack (this is clearly regular).

## 3.3 Reduction to Lower Orders

We are now ready to complete the reduction. Correctness is shown in Section 4. Let $A_{\mathtt{tt}}$ be the automaton accepting all stacks. In the following definition, a control state $(p_1, p_2)$ means that we are currently in control state $p_1$ and are aiming to empty the stack on reaching $p_2$, and the rules $\mathcal{R}_{\text{sim}}$ simulate all operations apart from $\text{push}_n$ and $\text{pop}_n$ directly, $\mathcal{R}_{\text{fin}}$ detect when the run is accepting, $\mathcal{R}_{\text{push}}$ follow the push branch of the tree decomposition, using tests to ensure the existence of the pop branch, and $\mathcal{R}_{\text{pop}}$ follow the pop branch of the tree decomposition, also using tests to check the existence of the push branch.

DEFINITION 3.4 ($P_{-1}$). *Given an $n$-PDA $P$ described by the tuple $(\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \{p_f\}, p_{\text{in}}, \gamma_{\text{in}})$ as well as families of automata $(A_{p_1,p_2})_{p_1,p_2 \in \mathcal{P}}$ and $\left(A^a_{p_1,p_2}\right)_{p_1,p_2 \in \mathcal{P}}$ we define an $(n-1)$-PDA with tests*

$$P_{-1} = (\mathcal{P}_{-1}, \Sigma, \Gamma, \mathcal{R}_{-1}, \mathcal{F}_{-1}, (p_{\text{in}}, p_f), \gamma_{\text{in}})$$

*where*

$$\begin{array}{rcl} \mathcal{P}_{-1} & = & \{(p_1, p_2) \mid p_1, p_2 \in \mathcal{P}\} \uplus \{f\} \\ \mathcal{R}_{-1} & = & \mathcal{R}_{sim} \cup \mathcal{R}_{fin} \cup \mathcal{R}_{push} \cup \mathcal{R}_{pop} \\ \mathcal{F}_{-1} & = & \{f\} \end{array}$$

*and we define*

- $\mathcal{R}_{sim}$ *is the set containing all rules of the form*

$$((p_1, p_2), \gamma, A_{\mathtt{tt}}) \xrightarrow{b} ((p'_1, p_2), o)$$

*for all* $(p_1, \gamma) \xrightarrow{b} (p'_1, o) \in \mathcal{R}$ *with* $o \notin \{push_n, pop_n\}$ *and* $p_2 \in \mathcal{P}$, *and*

- $\mathcal{R}_{fin}$ *is the set containing all rules of the form*

$$((p_1, p_2), \gamma, A_{\mathtt{tt}}) \xrightarrow{\varepsilon} (f, rew_\gamma)$$

*for all* $(p_1, \gamma) \xrightarrow{\varepsilon} (p_2, pop_n) \in \mathcal{R}$, *and*

- $\mathcal{R}_{push}$ *is the smallest set of rules containing all rules of the form*

$$((p_1, p_2), \gamma, A_{p,p_2}) \xrightarrow{\varepsilon} ((p'_1, p), rew_\gamma)$$

*for all* $(p_1, \gamma) \xrightarrow{\varepsilon} (p'_1, push_n) \in \mathcal{R}$ *and* $p, p_2 \in \mathcal{P}$, *and all rules of the form*

$$((p_1, p_2), \gamma, A^a_{p,p_2}) \xrightarrow{a} ((p'_1, p), rew_\gamma)$$

*for all* $(p_1, \gamma) \xrightarrow{\varepsilon} (p'_1, push_n) \in \mathcal{R}$ *and* $p, p_2 \in \mathcal{P}$, *and*

- $\mathcal{R}_{pop}$ *is the set containing all rules of the form*

$$\left((p_1, p_2), \gamma, A_{p'_1,p}\right) \xrightarrow{\varepsilon} ((p, p_2), rew_\gamma)$$

*for all* $(p_1, \gamma) \xrightarrow{\varepsilon} (p'_1, push_n) \in \mathcal{R}$ *and* $p, p_2 \in \mathcal{P}$ *and all rules of the form*

$$\left((p_1, p_2), \gamma, A^a_{p'_1,p}\right) \xrightarrow{a} ((p, p_2), rew_\gamma)$$

*for all* $(p_1, \gamma) \xrightarrow{\varepsilon} (p'_1, push_n) \in \mathcal{R}$ *and* $p, p_2 \in \mathcal{P}$.

In the next section, we show the reduction is correct.

LEMMA 3.1 (Correctness of $P_{-1}$).

$$Diagonal_a(P) \iff Diagonal_a(P_{-1})$$

To complete the reduction, we convert the HOPDA with tests into a HOPDA without tests.

LEMMA 3.2 (Reduction to Lower Orders). *For every $n$-PDA $P$ we can construct an $(n-1)$-PDA $P'$ such that*

$$Diagonal_a(P) \iff Diagonal_a(P') .$$

*Proof.* From Definition 3.4 ($P_{-1}$) and Lemma 3.1 (Correctness of $P_{-1}$), we obtain from $P$ an $(n-1)$-PDA with tests $P_{-1}$ satisfying the conditions of the lemma. To complete the proof, we invoke Theorem 3.2 (Removing Tests) to find $P'$ as required. □

## 4. Correctness of Reduction

This section is dedicated to the proof of Lemma 3.1 (Correctness of $P_{-1}$).

The idea of the proof is that each run of $P$ can be decomposed into a tree: each push$_n$ operation creates a node whose left child is the run up to the matching pop$_n$, and whose right child is the run after the matching pop$_n$. All other operations create a node with a single child which is the successor configuration.

Each branch of such a tree corresponds to a run of $P_{-1}$. To prove that $P_{-1}$ can output an unbounded number of $a$s we prove that any tree containing $m$ edges outputting $a$ must have a branch along which $P_{-1}$ would output $\log(m)$ $a$ characters. Thus, if $P$ can output an unbounded number of $a$ characters, so can $P_{-1}$.

### 4.1 Tree Decomposition of Runs

Given a run

$$\rho = c_0 \xrightarrow{b_1} c_1 \xrightarrow{b_2} \cdots \xrightarrow{b_m} c_m$$

of $P$ where each push$_n$ operation has a matching pop$_n$, we can construct a tree representation of $\rho$ inductively. That is, we define Tree$(c) = T[\varepsilon]$ for the single-configuration run $c$, and, when

$$\rho = c \xrightarrow{b} \rho'$$

where the first rule applied does not contain a push$_n$ operation, we have

$$\text{Tree}(\rho) = b\big[\text{Tree}\big(\rho'\big)\big]$$

and, when

$$\rho = c_0 \xrightarrow{\varepsilon} \rho_1 \xrightarrow{\varepsilon} \rho_2$$

with $c_1$ being the first configuration of $\rho_2$ and where the first rule applied in $\rho$ contains a push$_n$ operation, $c_0 = \langle p, s\rangle$ and $c_1 = \langle p', s\rangle$ for some $p, p', s$ and there is no configuration in $\rho_1$ of the form $\langle p'', s\rangle$, then

$$\text{Tree}(\rho) = \varepsilon[\text{Tree}(\rho_1), \text{Tree}(\rho_2)] .$$

An accepting run of $P$ has the form $\rho \xrightarrow{\varepsilon} c$ where $\rho$ has the property that all push$_n$ operations have a matching pop$_n$ and the final transition is a pop$_n$ operation to $c = \langle p, []_n\rangle$ for some $p \in \mathcal{F}$. Hence, we define the tree decomposition of an accepting run to be

$$\text{Tree}\left(\rho \xrightarrow{\varepsilon} c\right) = \varepsilon[\text{Tree}(\rho), T[\varepsilon]] .$$

### 4.2 Scoring Trees

In the above tree decomposition of runs, the tree branches at each instance of a push$_n$ operation. This mimics the behaviour of $P_{-1}$, which performs such branching non-deterministically. Hence, given a run $\rho$ of $P$, each branch of Tree$(\rho)$ corresponds to a run of $P_{-1}$.

We formalise this intuition in the following section. In this section, we assign scores to each subtree $T$ of Tree$(\rho)$. These scores correspond directly to the largest number of $a$ characters that $P_{-1}$ can output while simulating a branch of $T$.

Note, in the following definition, we exploit the fact that only nodes with exactly one child may have a label other than $\varepsilon$. We also give a general definition applicable to trees with out-degree larger than 2. This is needed in the simultaneous unboundedness section. For the moment, we only have trees with out-degree at most 2.

Let

$$\overline{b} = \begin{cases} 0 & b = \varepsilon \\ 1 & b = a \end{cases} \quad \text{and} \quad \overline{m} = \begin{cases} 0 & m = 0 \\ 1 & m > 0 \end{cases} .$$

Then, Score$(T)$ =

$$\begin{cases} 0 & T = T[\varepsilon] \\ \text{Score}(T_1) + \overline{b} & T = b[T_1] \\ \max\limits_{1 \le i \le m}\left(\text{Score}(T_i) + \overline{\sum\limits_{j \ne i} \text{Score}(T_j)}\right) & T = \varepsilon[T_1, \ldots, T_m] \end{cases}$$

We then have the following lemma for trees with out-degree 2.

LEMMA 4.1 (Minimum Scores). *Given a tree $T$ containing $m$ nodes labelled $a$, we have*

$$Score(T) \ge \log(m)$$

*Proof.* The proof is by induction over $m$. In the base case $m = 1$ and there is a single node $\eta$ in $T$ labelled $a$. By definition, the subtree $T'$ rooted at $\eta$ has Score$(T') = 1$. Since the score of a tree is bounded from below by the score of any of its subtrees, we have Score$(T) \ge \log(1)$ as required.

Now, assume $m > 1$. Find the smallest subtree $T'$ of $T$ containing $m$ nodes labelled $a$. We necessarily have either

1. $T' = a[T_1]$, or

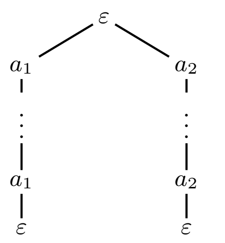


Figure 3: An example showing that following a single branch does not work for simultaneous unboundedness.

2. $T' = \varepsilon[T_1, T_2]$ where $T_1$ and $T_2$ each have at least one node each labelled $a$.

In case (1) we have by induction

$$\mathrm{Score}\big(T'\big) = 1 + \log(m-1) \geq \log(m)$$

In case (2) we have

$$\mathrm{Score}\big(T'\big) = \max\left(\begin{array}{c} \mathrm{Score}(T_1) + \overline{\mathrm{Score}(T_2)}, \\ \mathrm{Score}(T_2) + \overline{\mathrm{Score}(T_1)} \end{array}\right) .$$

We pick whichever of $T_1$ and $T_2$ has the most nodes labelled $a$. This tree has at least $\lceil m/2 \rceil$ nodes labelled $a$. Note, since both trees contain nodes labelled $a$, the right-hand side of the addition is always 1. Hence, we need to show

$$\log(\lceil m/2 \rceil) + 1 \geq \log(m)$$

which follows from

$$\begin{array}{c} \log(m) - \log(\lceil m/2 \rceil) = \log\Big(\frac{m}{\lceil m/2 \rceil}\Big) \\ \leq \\ \log\Big(\frac{m}{m/2}\Big) = \log(2) = 1 \ . \end{array}$$

By our choice of $T'$ we thus have $\mathrm{Score}(T) = \mathrm{Score}(T') \geq \log(m)$ as required. □

### 4.3 Completing the Proof

To complete the proof we show the following lemmas, whose proofs are given in the full version [19]. These lemmas simply formalise the connection between runs of $P$ and runs of $P_{-1}$.

LEMMA 4.2 (Scores to Runs). *Given an accepting run $\rho$ of $P$, if $Score(Tree(\rho)) = m$ then $a^m \in \mathcal{L}(P_{-1})$.*

LEMMA 4.3 ($P_{-1}$ to $P$). *If $Diagonal_a(P_{-1})$ then $Diagonal_a(P)$.*

## 5. Multiple Characters

We generalise the previous result to the full diagonal problem. Naïvely, the previous approach cannot work. Consider the HOPDA executing

$$\mathrm{push}_1^m; \mathrm{push}_n; \mathrm{pop}_1^m; \mathrm{pop}_n; \mathrm{pop}_1^m$$

where the first sequence of $\mathrm{pop}_1$ operations output $a_1$ and the second sequence output $a_2$.

The corresponding run trees are of the form given in Figure 3. In particular, $P_{-1}$ can only choose one branch, hence all runs of $P_{-1}$ produce a bounded number of $a_1$s or a bounded number of $a_2$s. They cannot be simultaneously unbounded.

For $P_{-1}$ to be able to output both an unbounded number of $a_1$ and $a_2$ characters, it must be able to output two branches of the tree. To this end, we define a notion of $\alpha$-branch HOPDA, which output trees with up to $\alpha$ branches. We then show that the reduction from $n$-PDA to $(n-1)$-PDA can be generalised to $\alpha$-branch HOPDA.

### 5.1 Branching HOPDA

We define $n$-PDA outputting trees with at most $\alpha$ branches, denoted $(n, \alpha)$-PDA. Note, an $n$-PDA that outputs a word is an $(n, 1)$-PDA. Indeed, any $(n, \alpha)$-PDA is also an $(n, \alpha')$-PDA whenever $\alpha \leq \alpha'$.

DEFINITION 5.1 ($(n, \alpha)$-PDA). *We define an* order-$n$ $\alpha$-branch pushdown automaton ($(n, \alpha)$-PDA) *to be given by a tuple $P = (\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \mathcal{F}, p_{\mathrm{in}}, \gamma_{\mathrm{in}}, \theta)$ where $\mathcal{P}$, $\Sigma$, $\Gamma$, $\mathcal{F}$, $p_{\mathrm{in}}$, and $\gamma_{\mathrm{in}}$ are as in HOPDA. The set of rules $\mathcal{R} \subseteq \bigcup_{1 \leq m \leq \alpha} \mathcal{P} \times \Gamma \times \Sigma \times Ops_n \times \mathcal{P}^m$ together with a mapping $\theta : \mathcal{P} \to \{1, \ldots, \alpha\}$ such that for all $(p, \gamma, b, o, p_1, \ldots, p_m) \in \mathcal{R}$ we have $\theta(p) \geq \theta(p_1) + \cdots + \theta(p_m)$.*

We use the notation $(p, \gamma) \xrightarrow{b} (p_1, \ldots, p_m, o)$ to denote a rule $(p, \gamma, b, o, p_1, \ldots, p_m) \in \mathcal{R}$. Intuitively, such a rule generates a node of a tree with $m$ children. The purpose of the mapping $\theta$ is to bound the number of branches that this tree may have. Hence, at each branching rule, the quota of branches is split between the different subtrees. The existence of such a mapping implies this information is implicit in the control states and an $(n, \alpha)$-PDA can only output trees with at most $\alpha$ branches.

From the initial configuration $c_0 = \langle p_{\mathrm{in}}, [\![\gamma_{\mathrm{in}}]\!]_n \rangle$ a run of an $(n, \alpha)$-PDA is a tree $T = (D, \lambda)$ whose nodes are labelled with $n$-PDA configurations, and generates an output tree $T' = (D, \lambda')$ whose nodes are labelled with symbols from the output alphabet. Precisely

- $\lambda(\varepsilon) = c_0$, and
- for a node $\eta$ with children $\eta_1, \ldots, \eta_m$ and $\lambda(\eta) = \langle p, s \rangle$ there is a rule $(p, \gamma) \xrightarrow{b} (p_1, \ldots, p_m, o)$ such that for all $1 \leq i \leq m$ we have $\lambda(\eta_i) = \langle p_i, s' \rangle$ where $\mathrm{top}_1(s) = \gamma$, $s' = o(s)$. Moreover we have $\lambda'(\eta) = b$.
- For all leaf nodes $\eta$ we have $\lambda'(\eta) = \varepsilon$.

The run is accepting if for all leaf nodes $\eta$ we have $\lambda(\eta) = \langle p, []_n \rangle$ and $p \in \mathcal{F}$. Let $\mathcal{L}(P)$ be the set of output trees of $P$.

Given an output tree $T$ we write $|T|_a$ to denote the number of nodes labelled $a$ in $T$. For an $(n, \alpha)$-PDA $P$, we define

$$\begin{array}{l} \mathrm{Diagonal}_{a_1, \ldots, a_\alpha}(P) = \\ \qquad\qquad \forall m. \exists T \in \mathcal{L}(P). \forall 1 \leq i \leq \alpha . |T|_{a_i} \geq m \ . \end{array}$$

## 6. Reduction For Simultaneous Unboundedness

Given an $(n, \alpha)$-PDA $P$ we construct an $(n-1, \alpha)$-PDA $P_{-1}$ such that

$$\mathrm{Diagonal}_{a_1, \ldots, a_\alpha}(P) \iff \mathrm{Diagonal}_{a_1, \ldots, a_\alpha}(P_{-1}) \ .$$

Moreover, we show $\mathrm{Diagonal}_{a_1, \ldots, a_\alpha}(P)$ is decidable for a $(0, \alpha)$-PDA (i.e. a regular automaton outputting an $\alpha$-branch tree) $P$.

For simplicity, we assume for all rules $(p, \gamma) \xrightarrow{b} (p_1, \ldots, p_m, o)$ if $m > 1$ then $o = \mathrm{rew}_\gamma$ (i.e. the stack is unchanged). Additionally we have $b = \varepsilon$.

We also make analogous assumptions to the single character case. That is, we assume $\Sigma = \{a_1, \ldots, a_\alpha, \varepsilon\}$ and use $b$ to range over $\Sigma$. Moreover, all rules of the form $(p, \gamma) \xrightarrow{b} (p', o)$ with $o = \mathrm{push}_n$ or $o = \mathrm{pop}_n$ have $b = \varepsilon$. Finally, we assume acceptance is by reaching a unique control state in $\mathcal{F}$ with an empty stack.

### 6.1 Some Intuition

We briefly sketch the intuition behind the algorithm. We illustrate the reduction from $(n, \alpha)$-PDA to $(n-1, \alpha)$-PDA in Figure 4.

- We begin with an $n$-PDA which we first interpret as an $(n, \alpha)$-PDA. This is possible because an $(n, \alpha)$-PDA can produce *at most* $\alpha$ branches. Thus, an $n$-PDA — which produces a single branch — is also a $(n, \alpha)$-PDA. We work with HOPDA producing $\alpha$ branches because, after each reduction step, we will need to output one branch for each character in $a_1, \ldots, a_\alpha$.

- We have an $(n, \alpha)$-PDA $P$ that outputs a tree with at most $\alpha$ branches. In Figure 4 we show part of a run tree with 2 branches. The $\text{push}_n$ and $\text{pop}_n$ operations are shown on the edges of the tree. Nodes are numbered to help identify them during the different transformations.
- We "decompose" this tree into another tree where the branches appearing after the $\text{pop}_n$ operations are hung from the same parent as their matching $\text{push}_n$. This is shown in the middle of Figure 4. Notice that this tree has an unbounded number of branches (it branches at each $\text{push}_n$). However, we know that the maximum out-degree of any of its nodes is $(\alpha + 1)$ since the source of a $\text{push}_n$-labelled edge has one child, and we add at most $\alpha$ extra children corresponding to the $\text{pop}_n$ on each of its at most $\alpha$ branches.
- We prove a generalisation of Lemma 4.1 (Minimum Scores) that shows a run tree with at least $m$ instances of a character $a$ has a branch with a score of at least $\log_{(\alpha+1)}(m)$. Thus, we need to select one branch for each $a$ we wish to output.
- We build an $(n-1, \alpha)$-PDA $P_{-1}$ that non-deterministically picks out the highest scoring branches for each $a$. This is shown on the right of Figure 4.

### 6.2 Branching HOPDA with Regular Tests

As before, we instrument our HOPDA with tests. Removing these tests requires a simple adaptation of Broadbent *et al.* [6].

DEFINITION 6.1 ($(n, \alpha)$-PDA with Tests). *Given a sequence of automata $A_1, \ldots, A_m$, an $(n, \alpha)$-PDA with tests is given by a tuple $P = (\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \mathcal{F}, p_{\text{in}}, \gamma_{\text{in}}, \theta)$ where $\mathcal{P}$, $\Sigma$, $\Gamma$, $\mathcal{F}$, $p_{\text{in}}$, $\gamma_{\text{in}}$ are as in HOPDA. The set of rules $\mathcal{R} \subseteq \bigcup_{1 \le m \le \alpha} \mathcal{P} \times \Gamma \times \{A_1, \ldots, A_m\} \times \Sigma \times Ops_n \times \mathcal{P}^m$ together with a mapping $\theta : \mathcal{P} \to \{1, \ldots, \alpha\}$ such that for all $(p, \gamma, A, b, o, p_1, \ldots, p_m) \in \mathcal{R}$ we have $\theta(p) \ge \theta(p_1) + \cdots + \theta(p_m)$.*

We use the notation $(p, \gamma, A) \xrightarrow{b} (p_1, \ldots, p_m, o)$ to denote a rule $(p, \gamma, A, b, o, p_1, \ldots, p_m) \in \mathcal{R}$.

From the initial configuration $c_0 = \langle p_{\text{in}}, [\![\gamma_{\text{in}}]\!]_n \rangle$ a run of an $(n, \alpha)$-PDA with tests is a tree $T = (D, \lambda)$ and generates an output tree $\rho = (D, \lambda')$ where

- $\lambda(\varepsilon) = c_0$, and
- for a node $\eta$ with children $\eta_1, \ldots, \eta_m$ and $\lambda(\eta) = \langle p, s \rangle$ there is a rule $(p, \gamma, A) \xrightarrow{b} (p_1, \ldots, p_m, o)$ such that $s \in \mathcal{L}(A)$ and for all $1 \le i \le m$ we have $\lambda(\eta_i) = \langle p_i, s' \rangle$ where $\text{top}_1(s) = \gamma$, and $s' = o(s)$. Moreover we have $\lambda'(\eta) = b$.
- For all leaf nodes $\eta$ we have $\lambda'(\eta) = \varepsilon$.

The run is accepting if for all leaf nodes $\eta$ we have $\lambda(\eta) = \langle p, []_n \rangle$ and $p \in \mathcal{F}$. Let $\mathcal{L}(P)$ be the set of output trees of $P$.

THEOREM 6.1 (Removing Tests). *[6, Theorem 3 (adapted)] For every $(n, \alpha)$-PDA with tests $P$, we can compute an $(n, \alpha)$-PDA $P'$ with $\mathcal{L}(P) = \mathcal{L}(P')$.*

The adapted proof of the above theorem is given in the full version [19].

### 6.3 Building The Automata

Previously we built automata $A_{p_1, p_2}$ to indicate that from $p_1$, the current top stack could be removed, arriving at $p_2$. This is fine for words, however, we now have $\alpha$-branch trees. It is no longer enough to specify a single control state: the top stack may be popped once on each branch of the tree, hence for a control state $p$ we need to recognise configurations with control state $p$ from which there is a run tree where the leaves of the trees are labelled with configurations with control states $p_1, \ldots, p_m$ and empty stacks. Moreover we need to recognise the set $O$ of characters output by the run tree. More precisely, for these automata we write

$$A^O_{p, p_1, \ldots, p_m}$$

where $\theta(p) \ge \theta(p_1) + \cdots + \theta(p_m)$ and $O \subseteq \{a_1, \ldots, a_\alpha\}$. We have $s \in \mathcal{L}\big(A^O_{p, p_1, \ldots, p_m}\big)$ iff there is a run tree $T$ with the root labelled $\langle p, [s]_n \rangle$ and $m$ leaf nodes labelled $\langle p_1, []_n \rangle, \ldots, \langle p_m, []_n \rangle$ respectively. Moreover, we have $a \in O$ iff the corresponding output tree $T'$ has $|T'|_a > 0$.

#### 6.3.1 Alternating HOPDA

To construct the required stack automata, we need to do reachability analysis of $(n, \alpha)$-PDA. We show that such analyses can be rephrased in terms of alternating higher-order pushdown systems (HOPDS), for which the required algorithms are already known [7]. Note, we refer to these machines as "systems" rather than "automata" because they do not output a language.

DEFINITION 6.2 (Alternating HOPDS). *An* alternating order-$n$ pushdown system *is a tuple $P = (\mathcal{P}, \Gamma, \mathcal{R})$ where $\mathcal{P}$ is a finite set of control states, $\Gamma$ is a finite stack alphabet, and*

$$\mathcal{R} \subseteq (\mathcal{P} \times \Gamma \times Ops_n \times \mathcal{P}) \cup \left(\mathcal{P} \times \Gamma \times 2^{\mathcal{P}}\right)$$

*is a set of transition rules.*

We write $(p, \gamma) \to (p, o)$ to denote $(p, \gamma, o, p) \in \mathcal{R}$ and $(p, \gamma) \to p_1, \ldots, p_m$ to denote $(p, \gamma, \{p_1, \ldots, p_m\}) \in \mathcal{R}$.

An run of an alternating HOPDS may split into several configurations, each of which must reach a target state. Hence, the branching of the alternating HOPDS mimics the branching of the $(n, \alpha)$-PDA. Given a set $C$ of configurations, we define $\text{Pre}^*_P(C)$ to be the smallest set $C'$ such that

$$\begin{aligned} C' \;=\;& C \,\cup \\ & \left\{ \langle p, s \rangle \;\middle|\; \begin{array}{c} (p, \gamma) \to (p', o) \in \mathcal{R} \,\wedge \\ \text{top}_1(s) = \gamma \,\wedge \\ \langle p', o(s) \rangle \in C' \end{array} \right\} \cup \\ & \left\{ \langle p, s \rangle \;\middle|\; \begin{array}{c} (p, \gamma) \to p_1, \ldots, p_m \in \mathcal{R} \,\wedge \\ \text{top}_1(s) = \gamma \,\wedge \\ \forall i. \langle p_i, s \rangle \in C' \end{array} \right\}. \end{aligned}$$

#### 6.3.2 Constructing the Tests

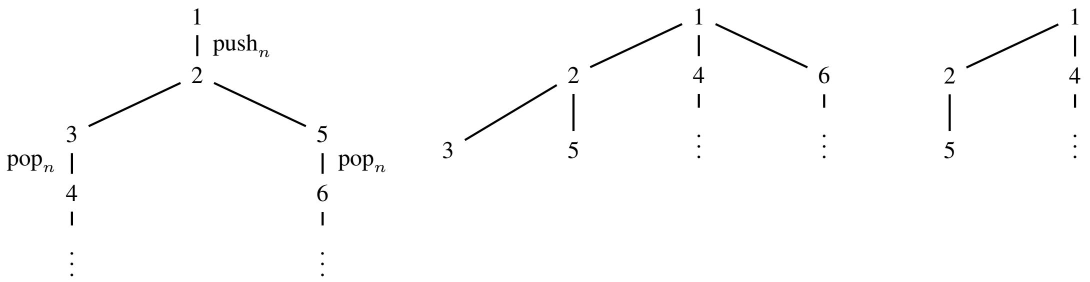
In order to use standard results to obtain $A^O_{p, p_1, \ldots, p_m}$ we construct an alternating HOPDS $P_\diamond$ and automaton $A$ such that checking $c \in \text{Pre}^*_{P_\diamond}(A)$ for a suitably constructed $c$ allows us to check whether $s \in \mathcal{L}\big(A^O_{p, p_1, \ldots, p_m}\big)$.

The alternating HOPDS $P_\diamond$ will mimic the branching of $P$ with alternating transitions[1] $(p, \gamma) \to p_1, \ldots, p_m$ of $P_\diamond$. It will maintain in its control states information about which characters have been output, as well as which control states should appear on the leaves of the branches. This final piece of information prevents all copies of the alternating HOPDS from verifying the same branch of $P$.

DEFINITION 6.3 ($P_\diamond$). *Given an $(n, \alpha)$-PDA $P$ described by the tuple $(\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \mathcal{F}, p_{\text{in}}, \gamma_{\text{in}})$, of $P$, we define*

$$P_\diamond = (\mathcal{P}_\diamond, \Gamma, \mathcal{R}_\diamond)$$

[1] We slightly alter the alternation rule from ICALP 2012 [7] by matching the top stack character as well as the control state. This is a benign alteration since it one can track the top of stack character in the control state.



(a) An $\alpha$-branch run tree of $P$

(b) The decomposition of the run tree

(c) An $\alpha$-branch subtree of (4b)

Figure 4: Illustrating the reduction steps.

*where*

$$\mathcal{P}_\diamond = \left\{ (p, O, p_1, \dots, p_m) \;\middle|\; \begin{array}{c} 1 \leq m \leq \alpha \wedge \\ O \subseteq \{a_1, \dots, a_\alpha\} \wedge \\ p_1, \dots, p_m \in \mathcal{P} \end{array} \right\}$$

*and $\mathcal{R}_\diamond$ is the set of rules containing, for each*

$$(p, \gamma) \xrightarrow{b} (p', o) \in \mathcal{R}$$

*all rules*

$$((p, O, p_1, \dots, p_i), \gamma) \rightarrow ((p_1, O \setminus \{b\}, p_1, \dots, p_i), o)$$

*and for each*

$$(p, \gamma) \xrightarrow{\varepsilon} (p_1, \dots, p_m, rew_\gamma) \in \mathcal{R}$$

*with $m > 1$ all alternating rules*

$$((p, O, p'_1, \dots, p'_i), \gamma) \rightarrow \begin{array}{c} (p_1, O_1, p^1_1, \dots, p^1_{i_1}), \\ \dots \\ (p_m, O_m, p^m_1, \dots, p^m_{i_m}) \end{array}$$

*where $p'_1, \dots, p'_i$ is a permutation of $p^1_1, \dots, p^1_{i_1}, \dots p^m_1, \dots, p^m_{i_m}$ and $O = O_1 \cup \dots \cup O_m$.*

In the above definition, the permutation condition ensures that the target control states are properly distributed amongst the newly created branches.

LEMMA 6.1. *We have $s \in \mathcal{L}(A^O_{p,p_1,\dots,p_m})$ iff*

$$\langle (p, O, p_1, \dots, p_m), [s]_n \rangle \in Pre^*_{P_\diamond}(A)$$

*where $A$ is such that*

$$\mathcal{L}(A) = \{ \langle (p, \emptyset, p), []_n \rangle \mid p \in \{p_1, \dots, p_m\} \} \ .$$

The proof of the above lemma is given in the full version [19].

It is known that $\text{Pre}^*_P(A)$ is computable for alternating HOPDS.

THEOREM 6.2. *[7, Theorem 1 (specialised)] Given an alternating HOPDS $P$ and a top-down automaton $A$, we can construct an automaton $A'$ accepting $Pre^*_P(A)$.*

Hence, we can now build $A^O_{p,p_1,\dots,p_m}$ from the control state $p$ and top-down automaton representation of $\text{Pre}^*_{P_\diamond}(A)$ since we can effectively translate from top-down to bottom-up stack automata.

## 6.4 Reduction to Lower Orders

We generalise our reduction to $(n, \alpha)$-PDA. Let $A_{\texttt{tt}}$ be the automata accepting all configurations. Note, in the following definition we allow all transitions (including branching) to be labelled by sets of output characters. To maintain our assumed normal form we have to replace these transitions using intermediate control states to ensure all branching transitions are labelled by $\varepsilon$ and all transitions labelled $O$ are replaced by a sequence of transitions outputting a single instance of each character in $O$.

The construction follows the intuition of the single character case, but with a lot more bookkeeping. Given an $(n, \alpha)$-PDA $P$ we define an $(n-1, \alpha)$-PDA with tests $P_{-1}$ such that $P$ satisfies the diagonal problem iff $P_{-1}$ also satisfies the diagonal problem. The main control states of $P_{-1}$ take the form

$$(p, p_1, \dots, p_m, O, B)$$

where $p, p_1, \dots, p_m$ are control states of $P$ and both $O$ and $B$ are sets of output characters. We explain the purpose of each of these components.

We will define $P_{-1}$ to generate up to $m$ branches of the tree decomposition of a run of $P$. In particular, for each of the characters $a \in \{a_1, \dots, a_\alpha\}$ there will be a branch of the run of $P_{-1}$ responsible for outputting "enough" of the character $a$ to satisfy the diagonal problem. Note that two characters $a$ and $a'$ may share the same branch. When a control state of the above form appears on a node of the run tree, the final component $B$ makes explicit which characters the subtree rooted at that node is responsible for generating in large numbers. Thus, the initial control state will have $B = \{a_1, \dots, a_\alpha\}$ since all characters must be generated from this node. However, when the output tree branches – i.e. a node has more than one child – the contents of $B$ will be partitioned amongst the children. That is, the responsibility of the parent to output enough of the characters in $B$ is divided amongst its children.

The remaining components play the role of a test $A^O_{p,p_1,\dots,p_m}$. That is, the current node is simulating the control state $p$ of $P$, and is required to produce $m$ branches, where the stack is emptied on each leaf and the control states appearing on these leaves are $p_1, \dots, p_m$. Moreover, the tree should output at least one of each character in $O$.

Note, $P_{-1}$ also has (external) tests of the form $A^O_{p,p_1,\dots,p_m}$ that it can use to make decisions, just like in the single character case. However, it also performs tests "online" in its control states. This is necessary because the tests were used to check what could have happened on branches not followed by $P_{-1}$. In the single character case, there was only one branch, hence $P_{-1}$ would uses tests to check all the branches not followed, and then continue down a single branch of the tree. In the multi-character case the situation is different. Suppose a subtree rooted at a given node was responsible for outputting enough of both $a_1$ and $a_2$. Amongst the possible children of this node we may select two children: one for outputting enough $a_1$ characters, and one for outputting enough $a_2$ characters. The alternatives not taken will be checked using tests as before. However, the child responsible for outputting $a_1$ may have also wanted to run a test on the child responsible for outputting $a_2$. Thus, as well as having to output enough $a_2$ characters, this latter child will also have to run the test required by the former. Thus, we have to build these tests into the control state. As a sanity condition we enforce $O \cap B = \emptyset$ since a branch outputting $a$ should never ask itself if it is able to produce at least one $a$.

We explain the rules of $P_{-1}$ intuitively. It will be beneficial to refer to the formal definition (below) while reading the explanations. The case for $\mathcal{R}_{\text{push}}$ is illustrated in Figure 5 since it covers most of the situations appearing in the other rules as well.

- The rules in $\mathcal{R}_{\text{init}}$ guess how many branches will be needed to output enough of each $a$. (This might be less than $\alpha$ since one branch might account for several characters.)
- The rules in $\mathcal{R}_{\text{fin}}$ check whether the run can be finished (always via a $pop_n$ since we are aiming for the empty stack). This is true if we only have one branch to complete (just reach $p'$) and we have no more characters that we're obliged to output.
- The rules in $\mathcal{R}_{\text{sim}}$ simulate a non-branching operation. They do this faithfully, simply passing along all information (updating $O$ if a character is output by the simulated transition).
- The rules in $\mathcal{R}_{\text{br}}$ are the first of the complicated rules. This is mainly a matter of notation. The reasoning behind the rules is that we're at a point where the tree splits into $l$ different branches. These have control states $p'_1, \ldots, p'_l$ respectively. We non-deterministically guess which of these branches should output which of the characters in $B$. Thus, we split $B$ into $B_1, \ldots, B_i$. This means we are exploring $i$ branches. Let $x_1, \ldots, x_i$ be the control states on these branches. The remaining branches we handle using tests on the stack. Let $y_1, \ldots, y_j$ be the control states appearing on these branches. We require that all of $p'_1, \ldots, p'_l$ are accounted for, so we assert that $p'_1, \ldots, p'_l$ is a permutation of $x_1, \ldots, x_i, y_1, \ldots, y_j$.

  Similarly, in the current subtree we are obliged to pop to leaf nodes containing the control states $p_1, \ldots, p_m$. We split these obligations between the branches we are exploring and those we are handling using tests. We use another permutation check to ensure the obligations have been distributed properly.

  Finally, we are required to output characters in $O$. We may also, in choosing a particular branch for a character $a$, need to output $a$ to account for instances appearing on a missed branch. Hence we also output $O'$ to account for these. We distribute the obligations $O$ and $O'$ amongst the different branches using $X_1, \ldots, X_i$ and $Y_1, \ldots, Y_j$.
- The rules in $\mathcal{R}_{\text{push}}$ and $\mathcal{R}_{\text{pop}}$ follow the same intuition as in the single character case, except we have the branching to deal with. In particular, at a push we have one branch corresponding to exploring what happens between the push and the corresponding pops, and a branch for each of the corresponding pops. We choose a selection of these branches to track with the HOPDA and a selection to handle using tests. The difference between $\mathcal{R}_{\text{push}}$ and $\mathcal{R}_{\text{pop}}$ is that the former explores the branch of the push using the HOPDA and the latter uses a test.

  In these rules, after the push we're in control state $p'$ and we guess that we will pop to control states $p'_1, \ldots, p'_l$. Hence we have a branch or a test to ensure that this happens. The remaining branches and tests are for what happens after the pops. The start from the states $p'_1, \ldots, p'_l$ and must, in total, pop to the original pop obligation $p_1, \ldots, p_m$. Hence, we distribute these tasks in the same way as the $\mathcal{R}_{\text{br}}$.

Before giving the formal definition, we summarise the discussion above by recalling the meaning of the various components. A control state $(p, p_1, \ldots, p_m, O, B)$ means we're currently simulating a node at control state $p$ that is required to produce $m$ branches terminating in control states $p_1, \ldots, p_m$ respectively, that the produced tree should output at least one of each character in $O$ and the entire subtree should output enough of each character in $B$ to satisfy the diagonal problem. In the definition below, the set $O'$ is the set of new single character output obligations produced when the automaton decides which branches to follow faithfully and which to test (for the output of at least one of each character). The sets $X_1, \ldots, X_i$ and $Y_1, \ldots, Y_j$ represent the partitioning of the single character output obligations amongst the tests and new branches.

The correctness of the reduction is stated after the definition. A discussion of the proof appears in Section 7.

DEFINITION 6.4 ($P_{-1}$). *Given an $(n, \alpha)$-PDA $P$ described by $(\mathcal{P}, \Sigma, \Gamma, \mathcal{R}, \{p_f\}, p_{\text{in}}, \gamma_{\text{in}}, \theta)$ and automata $A^{O}_{p,p_1,\ldots,p_m}$ for all $1 \leq m \leq \alpha$, $p, p_1, \ldots, p_m \in \mathcal{P}$, and $O \subseteq \{a_1, \ldots, a_\alpha\}$ we define an $(n-1, \alpha)$-PDA with tests*

$$P_{-1} = \left(\mathcal{P}_{-1}, \Sigma, \Gamma, \mathcal{R}_{-1}, \mathcal{F}_{-1}, p_{in}^{-1}, \gamma_{\text{in}}, \theta_{-1}\right)$$

*where $\mathcal{P}_{-1}$ is the set*

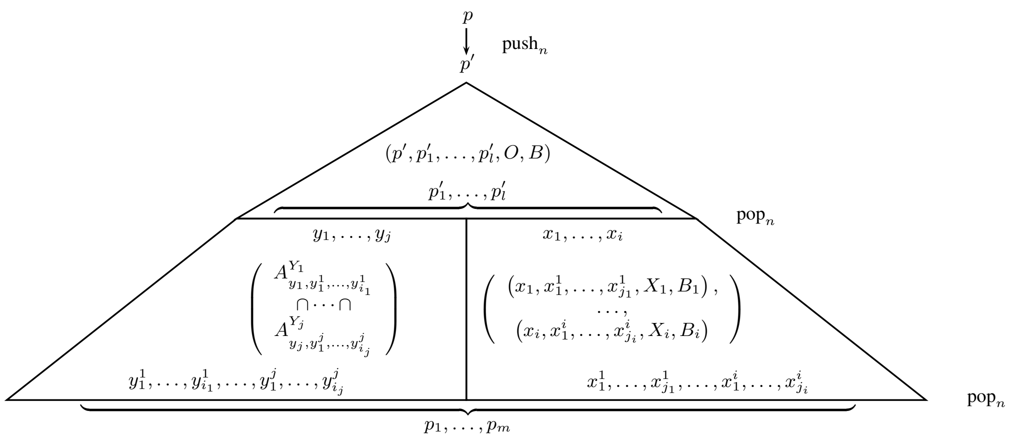
$$\left\{ (p, p_1, \ldots, p_m, O, B) \;\middle|\; \begin{array}{c} 1 \leq m \leq \alpha \wedge \\ p, p_1, \ldots, p_m \in \mathcal{P} \wedge \\ O, B \subseteq \{a_1, \ldots, a_\alpha\} \wedge \\ O \cap B = \emptyset \end{array} \right\} \uplus$$
$$\left\{p_{in}^{-1}, f\right\}$$

*and*

$$\begin{array}{rcl} \mathcal{R}_{-1} & = & \mathcal{R}_{init} \cup \mathcal{R}_{sim} \cup \mathcal{R}_{br} \cup \mathcal{R}_{fin} \cup \mathcal{R}_{push} \cup \mathcal{R}_{pop} \\ \mathcal{F}_{-1} & = & \{f\} \end{array}$$

*and $\theta_{-1}((p, p_1, \ldots, p_m, O, B)) = |B|$ and is $1$ for all other control states. We define the sets of rules, where in all cases, $p_1, \ldots, p_m \in \mathcal{P}$ and $O, O', B \subseteq \{a_1, \ldots, a_\alpha\}$, to be as follows:*

- *$\mathcal{R}_{init}$ is the set containing all rules of the form*
  $$\left(p_{in}^{-1}, \gamma_{\text{in}}\right) \xrightarrow{\varepsilon} \left((p_{\text{in}}, p_f, \ldots, p_f, \emptyset, \{a_1, \ldots, a_\alpha\}), rew_{\gamma_{\text{in}}}\right)$$
  *where $|p_f, \ldots, p_f| \leq \alpha$, and*
- *$\mathcal{R}_{fin}$ is the set containing all rules of the form*
  $$\left((p, p', \emptyset, B), \gamma, A_{\mathtt{tt}}\right) \xrightarrow{\varepsilon} (f, rew_\gamma)$$
  *for all $(p, \gamma) \xrightarrow{\varepsilon} (p', pop_n) \in \mathcal{R}$ and $B \subseteq \{a_1, \ldots, a_\alpha\}$, and*
- *$\mathcal{R}_{sim}$ is the set containing all rules of the form*
  $$\begin{tikzcd} ((p, p_1, \ldots, p_m, O, B), \gamma, A_{\mathtt{tt}}) \arrow[d, "\{b\} \cap B"] \\ ((p', p_1, \ldots, p_m, O \setminus \{b\}, B), o) \end{tikzcd}$$
  *for $(p, \gamma) \xrightarrow{b} (p', o) \in \mathcal{R}$, and $o \notin \{push_n, pop_n\}$, and*
- *$\mathcal{R}_{br}$ is the set containing all rules of the form*
  $$\begin{tikzcd} \left( (p, p_1, \ldots, p_m, O, B), \gamma, \begin{array}{c} A^{Y_1}_{y_1, y^1_1, \ldots, y^1_{i_1}} \\ \cap \cdots \cap \\ A^{Y_j}_{y_j, y^j_1, \ldots, y^j_{i_j}} \end{array} \right) \arrow[d, "O' \cap B"] \\ \left( \begin{array}{c} \left(x_1, x^1_1, \ldots, x^1_{j_1}, X_1, B_1\right), \\ \ldots, \\ \left(x_i, x^i_1, \ldots, x^i_{j_i}, X_i, B_i\right) \end{array}, rew_\gamma \right) \end{tikzcd}$$
  *where*
  $$(p, \gamma) \xrightarrow{\varepsilon} \left(p'_1, \ldots, p'_l, rew_\gamma\right) \in \mathcal{R}$$



$p$
push$_n$
$p'$
$(p', p'_1, \ldots, p'_l, O, B)$
$p'_1, \ldots, p'_l$
pop$_n$
$y_1, \ldots, y_j$
$x_1, \ldots, x_i$
$\begin{pmatrix} A^{Y_1}_{y_1, y^1_1, \ldots, y^1_{i_1}} \\ \cap \cdots \cap \\ A^{Y_j}_{y_j, y^j_1, \ldots, y^j_{i_j}} \end{pmatrix}$
$\begin{pmatrix} (x_1, x^1_1, \ldots, x^1_{j_1}, X_1, B_1), \\ \ldots, \\ (x_i, x^i_1, \ldots, x^i_{j_i}, X_i, B_i) \end{pmatrix}$
$y^1_1, \ldots, y^1_{i_1}, \ldots, y^j_1, \ldots, y^j_{i_j}$
$x^1_1, \ldots, x^1_{j_1}, \ldots, x^i_1, \ldots, x^i_{j_i}$
pop$_n$
$p_1, \ldots, p_m$

Figure 5: Illustrating the rules in $\mathcal{R}_{\text{push}}$.

*and $p'_1, \ldots p'_l$ is a permutation of*

$$x_1, \ldots, x_i, y_1, \ldots, y_j$$

*and $p_1, \ldots, p_m$ is a permutation of*

$$x^1_1, \ldots, x^1_{j_1}, \ldots x^i_1, \ldots, x^i_{j_i} y^1_1, \ldots, y^1_{i_1}, \ldots y^j_1, \ldots, y^j_{i_j}$$

*and*

$$O \cup O' = X_1 \cup \cdots \cup X_i \cup Y_1 \cup \cdots \cup Y_j$$

*and $B = B_1 \cup \cdots \cup B_i$.*

- *$\mathcal{R}_{push}$ is the set containing all rules of the form*

$$\begin{pmatrix} (p, p_1, \ldots, p_m, O, B), \gamma, & \begin{matrix} A^{Y_1}_{y_1, y^1_1, \ldots, y^1_{i_1}} \\ \cap \cdots \cap \\ A^{Y_j}_{y_j, y^j_1, \ldots, y^j_{i_j}} \end{matrix} \end{pmatrix}$$

$$\Big\downarrow\ O' \cap B$$

$$\begin{pmatrix} \begin{matrix} (p', p'_1, \ldots, p'_l, X, B_0), \\ (x_1, x^1_1, \ldots, x^1_{j_1}, X_1, B_1), \\ \ldots, \\ (x_i, x^i_1, \ldots, x^i_{j_i}, X_i, B_i) \end{matrix} & , rew_\gamma \end{pmatrix}$$

*where*

$$(p, \gamma) \xrightarrow{\varepsilon} (p', push_n)$$

*and $p'_1, \ldots p'_l$ is a permutation of*

$$x_1, \ldots, x_i, y_1, \ldots, y_j$$

*and $p_1, \ldots, p_m$ is a permutation of*

$$x^1_1, \ldots, x^1_{j_1}, \ldots x^i_1, \ldots, x^i_{j_i} y^1_1, \ldots, y^1_{i_1}, \ldots y^j_1, \ldots, y^j_{i_j}$$

*and*

$$O \cup O' = X \cup X_1 \cup \cdots \cup X_i \cup Y_1 \cup \cdots \cup Y_j$$

*and $B = B_0 \cup \cdots \cup B_i$.*

- *we have $\mathcal{R}_{pop}$ is the set containing all rules of the form*

$$\begin{pmatrix} (p, p_1, \ldots, p_m, O, B), \gamma, & \begin{matrix} A^{Y}_{p', p'_1, \ldots, p'_l} \cap \\ A^{Y_1}_{y_1, y^1_1, \ldots, y^1_{i_1}} \\ \cap \cdots \cap \\ A^{Y_j}_{y_j, y^j_1, \ldots, y^j_{i_j}} \end{matrix} \end{pmatrix}$$

$$\Big\downarrow\ O' \cap B$$

$$\begin{pmatrix} \begin{matrix} (x_1, x^1_1, \ldots, x^1_{j_1}, X_1, B_1), \\ \ldots, \\ (x_i, x^i_1, \ldots, x^i_{j_i}, X_i, B_i) \end{matrix} & , rew_\gamma \end{pmatrix}$$

*where*

$$(p, \gamma) \xrightarrow{\varepsilon} (p', push_n)$$

*and $p'_1, \ldots p'_l$ is a permutation of*

$$x_1, \ldots, x_i, y_1, \ldots, y_j$$

*and $p_1, \ldots, p_m$ is a permutation of*

$$x^1_1, \ldots, x^1_{j_1}, \ldots x^i_1, \ldots, x^i_{j_i} y^1_1, \ldots, y^1_{i_1}, \ldots y^j_1, \ldots, y^j_{i_j}$$

*and*

$$O \cup O' = Y \cup X_1 \cup \cdots \cup X_i \cup Y_1 \cup \cdots \cup Y_j$$

*and $B = B_1 \cup \cdots \cup B_i$.*

In Section 7 we show that the reduction is correct.

LEMMA 6.2 (Correctness of $P_{-1}$).

$$Diagonal_{a_1, \ldots, a_\alpha}(P) \iff Diagonal_{a_1, \ldots, a_\alpha}(P_{-1})$$

To complete the reduction, we convert the $(n, \alpha)$-PDA with tests into a $(n, \alpha)$-PDA without tests.

LEMMA 6.3 (Reduction to Lower Orders). *For every $(n, \alpha)$-PDA $P$ we can build an order-$(n-1)$ $\alpha$-branch HOPDA $P'$ such that*

$$Diagonal_{a_1, \ldots, a_\alpha}(P) \iff Diagonal_{a_1, \ldots, a_\alpha}(P') .$$

*Proof.* From Definition 6.4 ($P_{-1}$) and Lemma 6.2 (Correctness of $P_{-1}$), we obtain from $P$ an $(n-1, \alpha)$-PDA with tests $P_{-1}$

satisfying the conditions of the lemma. To complete the proof, we invoke Theorem 6.1 (Removing Tests) to find $P'$ as required. □

We show correctness of the reduction in Section 7. First we show that we have decidability once we have reduced to order-0.

### 6.5 Decidability at Order-0

We show that the problem becomes decidable for a 0-PDA $P$. This is essentially a finite state machine and we can linearise the trees generated by saving the list of states that have been branched to in the control state. After one branch has completed, we run the next in the list, until all branches have completed. Hence, a tree of $P$ becomes a run of the linearised 0-PDA, and vice-versa. Since each output tree has a bounded number of branches, the list length is bounded. Thus, we convert $P$ into a finite state word automaton, for which the diagonal problem is decidable. Note, this result can also be obtained from the decidability of the diagonal problem for pushdown automata. The details are given in the full version [19].

### 6.6 Decidability of The Diagonal Problem

THEOREM 6.3 (Decidability of the Diagonal Problem). *For an $n$-PDA $P$ and output characters $a_1, \ldots, a_\alpha$, it is decidable whether Diagonal$_{a_1,\ldots,a_\alpha}(P)$.*

*Proof.* We first interpret $P$ as an $(n, \alpha)$-PDA and then construct via Lemma 6.3 (Reduction to Lower Orders) an $(n-1, \alpha)$-PDA $P'$ such that Diagonal$_{a_1,\ldots,a_\alpha}(P)$ iff Diagonal$_{a_1,\ldots,a_\alpha}(P')$. We repeat this step until we have an $(0, \alpha)$-PDA. Then, from decidability at order-0 we obtain decidability as required. □

## 7. Correctness for Simultaneous Unboundedness

In this section we prove Lemma 6.2 (Correctness of $P_{-1}$). The proof follows the same outline as the single character case. To show there is a run with at least $m$ of each character, we take via Lemma 7.1 (Section 7.2), $m' = (\alpha + 1)^m$, and a run of $P$ outputting at least this many of each character. Then from Lemma 7.2 (Section 7.3) a run of $P_{-1}$ outputting at least $m$ of each character as required. The other direction is shown in Lemma 7.3 (Section 7.3).

We first generalise our tree decomposition and notion of scores. We then show that every $\alpha$-branch subtree of a tree decomposition generates a run tree of $P_{-1}$ matching the scores of the tree. Finally we prove the opposite direction.

### 7.1 Tree Decomposition of Output Trees

Given an output tree $T$ of $P$ where each push$_n$ operation has a matching pop$_n$ on all branches, we can construct a decomposed tree representation of the run inductively as follows. We define Tree$(T[\varepsilon]) = T[\varepsilon]$ and, when

$$T = b[T_1, \ldots, T_m]$$

where the rule applied at the root does not contain a push$_n$ operation, we have

$$\text{Tree}(T) = b[\text{Tree}(T_1), \ldots, \text{Tree}(T_m)] \ .$$

In the final case, let

$$T = \varepsilon\big[T'\big]$$

where the rule applied at the root contains a push$_n$ operation and the corresponding pop$_n$ operations occur at nodes $\eta_1, \ldots, \eta_m$.

Note, if the output trees had an arbitrary number of branches, $m$ may be unbounded. In our case, $m \leq \alpha$, without which our reduction would fail: $P_{-1}$ would be unable to accurately count the number of pop$_n$ nodes. In fact, our trees would have unbounded out degree and Lemma 4.1 (Minimum Scores) would not generalise.

Let $T_1, \ldots, T_m$ be the output trees rooted at $\eta_1, \ldots, \eta_m$ respectively and let $T'$ be $T$ with these subtrees removed. Observe all branches of $T$ are cut by this operation since the push$_n$ must be matched on all branches. We define

$$\text{Tree}(T) = \varepsilon\big[\text{Tree}\big(T'\big), \text{Tree}(T_1), \ldots, \text{Tree}(T_m)\big] \ .$$

An accepting run of $P$ has an extra pop$_n$ operation at the end of each branch leading to the empty stack. Let $T'$ be the tree obtained by removing the final pop$_n$-induced edge leading to the leaves of each branch. The tree decomposition of an accepting run is

$$\text{Tree}(T) = \varepsilon\big[\text{Tree}\big(T'\big), T[\varepsilon], \ldots, T[\varepsilon]\big]$$

where there are as many $T[\varepsilon]$ as there are leaves of $T$.

Notice that our trees have out-degree at most $(\alpha + 1)$.

### 7.2 Scoring Trees

We score branches in the same way as the single character case. We simply define Score$_a(\rho)$ to be Score$(\rho)$ when $a$ is considered as the only output character (all others are replaced with $\varepsilon$).

We have to slightly modify our minimum score lemma to accommodate the increased out-degree of the nodes in the trees.

LEMMA 7.1 (Minimum Scores). *Given a tree $T$ with maximum out-degree $(\alpha + 1)$, containing, for each $a \in \{a_1, \ldots, a_\alpha\}$, at least $m$ nodes labelled $a$, for each $a \in \{a_1, \ldots, a_\alpha\}$ we have*

$$\mathit{Score}_a(T) \geq \log_{(\alpha+1)}(m)$$

*Proof.* This is a simple extension of the proof of Lemma 4.1 (Minimum Scores). We simply replace the two-child case with a tree with up to $(\alpha + 1)$ children. In this case, we have to use $\log_{(\alpha+1)}$ rather than $\log$ to maintain the lemma. □

### 7.3 Completing the proof

As in the single character case, we complete the proof with the following two lemmas, shown in the full version [19]. As before, these lemmas simply formalise the fact that $P_{-1}$ runs along branches of a tree decomposition of a run of $P$.

LEMMA 7.2 (Scores to Runs). *Given an accepting output tree $\rho$ of $P$, if for all $a \in \{a_1, \ldots, a_\alpha\}$ we have Score$_a$(Tree$(\rho)$) $\geq m$, then $\exists T \in \mathcal{L}(P_{-1})$ with $|T|_a \geq m$ for all $a \in \{a_1, \ldots, a_\alpha\}$.*

LEMMA 7.3 ($P_{-1}$ to $P$). *We have*

$$\mathit{Diagonal}_{a_1,\ldots,a_\alpha}(P_{-1}) \Rightarrow \mathit{Diagonal}_{a_1,\ldots,a_\alpha}(P) \ .$$

## 8. Conclusions

We have shown, using a recent result by Zetzsche, that the downward closures of languages defined by HOPDA are computable. We believe this to be a useful foundational result upon which new analyses may be based. Our result already has several immediate consequences, including separation by piecewise testability and asynchronous parameterised systems.

Regarding the complexity of the approach. We are unaware of any complexity bounds implied by Zetzsche's techniques. Due to the complexity of the reachability problem for HOPDA, the test automata may be a tower of exponentials of height $n$ for HOPDA of order $n$. These test automata are built into the system before proceeding to reduce to order $(n-1)$. Thus, we may reach a tower of exponentials of height $O(n^2)$.

A natural next step is to consider collapsible pushdown systems, which are equivalent to recursion schemes (without the safety constraint). However, it is not currently clear how to generalise our techniques due to the non-local behaviour introduced by collapse. We may also try to adapt our techniques to a higher-order version of BS-automata [3], which may be used, e.g., to check boundedness of resource usage for higher-order programs.

## References

[1] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. Safety is not a restriction at level 2 for string languages. In *FOSSACS*, 2005.

[2] A. V. Aho. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968.

[3] M. Bojanczyk. Beyond omega-regular languages. In *STACS*, 2010.

[4] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, 2005.

[5] C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *CSL*, 2013.

[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] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-shore: a collapsible approach to higher-order verification. In *ICFP*, 2013.

[9] B. Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, 44:178–186, 1991.

[10] A. Cyriac, P. Gastin, and K. N. Kumar. MSO decidability of multi-pushdown systems via split-width. In *CONCUR*, 2012.

[11] W. Czerwinski and W. Martens. A note on decidable separability by piecewise testable languages. *CoRR*, abs/1410.1042, 2014. URL `http://arxiv.org/abs/1410.1042`.

[12] J. Esparza and P. Ganty. Complexity of pattern-based verification for multithreaded programs. In *POPL*, 2011.

[13] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL*, 2000.

[14] J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.

[15] M. Hague. Saturation of concurrent collapsible pushdown systems. In *FSTTCS*, 2013.

[16] M. Hague. Senescent ground tree rewrite systems. In *CSL-LICS*, 2014.

[17] M. Hague and A. W. Lin. Synchronisation- and reversal-bounded analysis of multithreaded programs with counters. In *CAV*, 2012.

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

[19] M. Hague, J. Kochems, and C. L. Ong. Unboundedness and downward closures of higher-order pushdown automata. *CoRR*, abs/1507.03304, 2015. URL `http://arxiv.org/abs/1507.03304`.

[20] L. Haines. On free monoids partially ordered by embedding. *J. Combinatorial Theory*, 6:9498, 1969.

[21] V. Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *LICS*, 2009.

[22] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FOSSACS*, 2002.

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

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

[25] 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.

[26] N. Kobayashi and A. Igarashi. Model-checking higher-order programs with recursive types. In *ESOP*, 2013.

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

[28] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *FMSD*, 35(1):73–97, 2009.

[29] P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, 2011.

[30] A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 15:1170–1174, 1976.

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

[32] P. Parys. On the significance of the collapse operation. In *LICS*, 2012.

[33] V. Penelle. Rewriting higher-order stack trees. In *CSR*, 2015.

[34] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[35] S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. In *POPL*, 2014.

[36] A. Seth. Games on higher order multi-stack pushdown systems. In *RP*, 2009.

[37] S. L. Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, 2011.

[38] S. L. Torre, A. Muscholl, and I. Walukiewicz. Safety of parametrized asynchronous shared-memory systems is almost always decidable. In *CONCUR*, 2015. To appear.

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

[40] J. van Leeuwen. Effective constructions in well-partially-ordered free monoids. *Discrete Mathematics*, 21(3):237252, 1978.

[41] G. Zetzsche. An approach to computing downward closures. In *ICALP*, 2015.