Collapsible Pushdown Automata and Recursion Schemes

Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong and Olivier Serre

[1] Royal Holloway, University of London
[2] DIMAP and Department of Computer Science, University of Warwick
[3] Department of Computer Science, University of Oxford
[4] IRIF, CNRS & Université Paris Diderot – Paris 7

Abstract

We consider recursion schemes (not assumed to be *homogeneously typed*, and hence not necessarily *safe*) and use them as generators of (possibly infinite) ranked trees. A recursion scheme is essentially a finite typed deterministic term rewriting system that generates, when one applies the rewriting rules *ad infinitum*, an infinite tree, called its *value tree*. A fundamental question is to provide an equivalent description of the trees generated by recursion schemes by a class of machines.

In this paper we answer this open question by introducing *collapsible pushdown automata* (CPDA), which are an extension of deterministic (higher-order) pushdown automata. A CPDA generates a tree as follows. One considers its transition graph, unfolds it and contracts its silent transitions, which leads to an infinite tree which is finally node labelled thanks to a map from the set of control states of the CPDA to a ranked alphabet.

Our contribution is to prove that these two models, higher-order recursion schemes and collapsible pushdown automata, are equi-expressive for generating infinite ranked trees. This is achieved by giving effective transformations in both directions.

1 Introduction

This paper establishes the equivalence of two models: *higher-order recursion schemes* and *collapsible pushdown automata*. A recursion scheme is a simply-typed term rewriting system. Deterministic recursion schemes can be viewed naturally as generators of possibly infinite trees. Collapsible pushdown automata (CPDA) are an extension of higher-order pushdown automata, and they naturally induce a transition graph. An infinite ranked tree can be constructed by first unfolding such a transition graph and then contracting the silent transitions. Applying this construction to CPDA defines a family of ranked trees which coincides with the family of ranked trees generated from higher-order recursion schemes.

Recursive Applicative Program Schemes

Recursion schemes have a long and rich history[1]. They go back to Nivat's *recursive applicative program schemes* [52], which correspond to order-1 recursion schemes in our sense, and to Garland and Luckham's *monadic recursion schemes* [31]. According to Nivat, a recursive applicative program scheme is a finite system of equations of the form $F_i(x_1, \ldots, x_{n_i}) = p_i$, where each x_j is an order-0 variable and p_i is an order-0 term constructed from the non-terminal symbols F_i, terminal symbols and the variables $x_1, \ldots, x_{n_i}$. A *program* is then a program scheme together with an *interpretation* in some domain. The least fixed point of the function defined by the rewriting rules of a program scheme gives a possibly infinite term

[1] De Miranda's thesis [27], among others, contains an account of the history. See also [55]

tree over the terminals alphabet, known as the value of the program in the *free / Hebrand interpretation*; applying the interpretation to this infinite term gives the *value* of the program. Thus the program scheme gives the uninterpreted syntax tree of some functional program that is then fully specified owing to the interpretation. For example, the term $\mathit{if}(\mathit{eq}(1,0),2,3)$ has the value 3 under the natural interpretation of *if*, *eq*, and the natural numbers.

Nivat also introduced a notion of equivalence: two program schemes are equivalent just if they compute the same function under *every* interpretation. Courcelle and Nivat [22] showed that two program schemes are equivalent if and only if they generate the same infinite term tree, thus underlining the importance of studying the tree generated by a scheme. Following the work of Courcelle [19, 20], the equivalence problem for program schemes is inter-reducible to the problem of language equivalence for deterministic pushdown automata (DPDA). The question of the decidability of the latter was first posed in the 1960s. It was only settled, positively, by Sénizergues in 1997 [66, 67], which therefore also established the decidability of the program scheme equivalence problem.

Extension of Schemes to Higher Orders

In Nivat's program scheme, the non-terminals and the variables are restricted to order 1 and 0 respectively. It follows that they are not suited to model higher-order recursive programs. A major theme in the late 1970s was the extension of program schemes to higher orders [36, 23, 24, 28, 29].

In an influential paper [25], Damm introduced *level-n λ-schemes*, extending the work of Courcelle and Nivat. Damm's schemes coincide with the *safe* fragment of the recursion schemes, which we will define later in the paper. It is important to note that so far there was no known model of automata equi-expressive with Damm's schemes; in particular, there was no known reduction of the equivalence problem for schemes to a language equivalence problem for (some model of) automata.

Later, Damm and Goerdt [25, 26] considered the word languages generated by level-n λ-schemes, and showed that they coincide with a hierarchy introduced earlier by Maslov [48, 49]. To define his hierarchy, Maslov introduced *higher-order pushdown automata* (Higher-order PDA); he also gave an equivalent definition of the hierarchy in terms of *higher-order indexed grammars*.

Higher-Order Recursion Schemes as Generators of Infinite Structures

Since the late 1990s, motivated mainly by applications to program verification, there has been a strong and sustained interest in infinite structures that admit finite descriptions; see [5] for an overview. The central question, given a class of such structures, is to find the most expressive logic for which model checking is decidable. Of course decidability here is a trade-off between richness of the structure and expressivity of the logic.

Of special interest are tree-like structures. Higher-order PDA as a generating device for possibly infinite labelled ranked trees were first studied by Knapik, Niwiński and Urzyczyn [40]. As in the case of word languages, an infinite hierarchy of trees is defined according to the order of the generating higher-order PDA; lower orders of the hierarchy are well-known classes of trees: orders 0, 1 and 2 are respectively the regular [59], algebraic [21] and hyperalgebraic trees [39]. Knapik *et al.* considered another method of generating such trees, namely, by higher-order (deterministic) recursion schemes that satisfy the *safety* constraint. A major result of their work is the equi-expressivity of both methods as tree generators. In particular it implies

that the equivalence problem for higher-order *safe* recursion schemes is inter-reducible to the problem of language equivalence for deterministic higher-order PDA.

An alternative approach was developed by Caucal [17] who introduced two infinite hierarchies, one consisting of infinite trees and the other of infinite graphs, defined by mutually recursive maps: unfolding which transforms graphs to trees, and inverse rational mapping (or MSO-interpretation [16]) which transforms trees to graphs. He showed that the tree hierarchy coincides with the trees generated by safe recursion schemes.

However, the fundamental question open since the early 1980s of finding a class of automata that characterises the expressivity of higher-order recursion schemes was left open. Indeed, the results of Damm and Goerdt, as well as those of Knapik *et al.* may only be viewed as attempts to answer the question as they both had to impose the same syntactic constraints on recursion schemes, called *derived types* and *safety* respectively, in order to establish their results.

A partial answer was later obtained by Knapik, Niwiński, Urzyczyn and Walukiewicz. They proved that order-2 homogeneously-typed, but not necessarily safe, recursion schemes are equi-expressive with a variant class of order-2 pushdown automata called *panic automata* [41].

Finally, we gave a complete answer to the question in an extended abstract [34]. For this, we introduced a new kind of higher-order pushdown automata, which generalises *pushdown automata with links* [3], or equivalently panic automata, to all finite orders, called *collapsible pushdown automata* (CPDA), in which every symbol in the stack has a link to a (necessarily lower-ordered) stack situated somewhere below it. A major result of [34] and of the present paper is that for every $n \geq 0$, order-n recursion schemes and order-n CPDA are equi-expressive as generators of trees.

Decidability of Monadic Second Order Logic

This quest of finding an alternative description of those trees generated by recursion schemes was led in parallel with the study of the decidability of the model-checking problem for the monadic second order logic (MSO) and the modal μ-calculus (see [72, 4, 32, 30] for background about these logics and connections with finite automata and games).

The decidability of the MSO theories of trees generated by safe recursion schemes of all finite orders was established by Knapik, Niwiński and Urzyczyn [40] and independently by Caucal [17] who proved, additionally, the MSO decidability of the associated graph hierarchy. The decidability result was first extended to possibly unsafe recursion schemes of order 2 by Knapik *et al.* [41] and Aehlig *et al.* [3] independently. The former group introduced panic automata and proved its equi-expressivity with the class of recursion schemes; the latter introduced an essentially equivalent automata model called pushdown automata with links.

In 2006, Ong established the MSO decidability of trees generated by recursion schemes of all finite orders [53]: he proved that the problem is n-EXPTIME complete. The result was obtained using techniques from innocent game semantics [35]; it does not rely on an equivalent automata model for generating trees.

A different, automata-theoretic, proof of Ong's decidability result was subsequently obtained by Hague, Murawski, Ong and Serre [34]. Thanks to the equi-expressivity between recursion schemes and CPDA, and the well-known connections between MSO model checking for trees and parity games, they show that the model-checking problem for recursion schemes is inter-reducible to the problem of determining the winner of a parity game played over the transition graph associated with a CPDA. Their work extends the techniques and results of (higher-order) pushdown games, including those of Walukiewicz [73], Cachat [13] (see also [14] for a comprehensive study on higher-order pushdown games) and Knapik *et al.* [41]. These

techniques have since been extended by Broadbent, Carayol, Ong and Serre to establish the closure of recursion schemes under MSO markings [9], and more recently by Carayol and Serre to prove that recursion schemes enjoy the effective MSO selection property [15].

Following initial ideas in [1] and [43], Kobayashi and Ong gave yet another proof of Ong's decidability result. Their proof [46] consists in showing that, given a recursion scheme and an MSO formula or an equivalent property, one can construct an intersection type system such that the scheme is typable in the type system if and only if the property is satisfied by the scheme. Typability is then reduced to solving a parity game.

In [61, 63], Salvati and Walukiewicz used Krivine machines [47] to represent the rewrite sequences of terms of the λY-calculus, a formalism equivalent to higher-order recursion schemes. A Krivine machine computes the weak head normal form of a λY-term using explicit substitutions. The MSO decidability for recursion schemes was then obtained by solving parity games played over the configurations of a Krivine machine. In [62, 64] they also provide a translation from recursion schemes to CPDA which is very close to the translation independently obtained by Carayol and Serre in [15]. Also note that in both of these translations the authors remark that if the original recursion scheme is safe the CPDA that is obtained can safely be transformed into a higher-order pushdown automaton (*i.e.* all collapse operations can be replaced by a standard popping); this was actually previously established by Blum in [6] and by Broadbent in his PhD thesis[8, chapter 3] for the translation we provide in this paper (as given in its conference version [34]). Also remark that neither [15] nor [64] provide the translation back from CPDA to schemes.

Let us stress that even if the proof of the translation from schemes to CPDA we give here is longer than the ones in [15, 64], it makes use of a richer higher-level concept (namely traversals) which we believe is worth knowing as it gives a deep insight on how scheme evaluation can be understood.

Structure of this paper

In this paper we present in full a proof of the equi-expressivity result which was first sketched in [34]. Owing to the length of this presentation, full proofs of the results therein on games played on the transition graphs of CPDA will be presented elsewhere.

The paper is organised as follows. Sections 2 and 3 introduce the main concepts, recursion schemes and CPDA respectively, together with examples. In Section 4 we state our main result. Then in Section 5 we give a transformation from CPDA to recursion schemes. The key idea is to associate a finite ground term with a given configuration of a CPDA and to provide rewriting rules for those terms that can simulate transitions of the CPDA. This gives rise to a transition system over finite ground terms that is isomorphic to the transition graph of the CPDA. The final step consists in simulating this transition system by an appropriate recursion scheme. Finally Section 6 gives the transformation in the other direction. For this we consider an intermediate object, the traversal tree of the recursion scheme, which turns out to be equivalent to the tree generated by the scheme. We then use the traversal tree to design an equivalent CPDA that computes paths in the traversal tree.

2 Recursion schemes

2.1 Types and terms

Types are generated by the grammar $A ::= o \mid A \to A$. Every type $A \neq o$ can be written

uniquely as $A_1 \to (A_2 \to \cdots \to (A_n \to o)\cdots)$, for some $n \geq 1$ which is called its *arity*; the *ground* type o has arity 0. We follow the convention that arrows associate to the right, and simply write $A_1 \to A_2 \to \cdots \to A_n \to o$, which we sometimes abbreviate to $(A_1, \ldots, A_n, o)$. The ***order*** of a type measures the nesting depth on the left of $\to$. We define $ord(o) = 0$ and $ord(A_1 \to A_2) = \max(ord(A_1) + 1, ord(A_2))$. Thus $ord(A_1 \to \ldots \to A_n \to o) = 1 + \max\{ord(A_i) \mid 1 \leq i \leq n\}$. For example, $ord(o \to o \to o \to o) = 1$ and $ord(((o \to o) \to o) \to o) = 3$.

Let Σ be a ***ranked alphabet*** *i.e.* each Σ-symbol f has an arity $ar(f) \geq 0$ which determines its type $\underbrace{o \to \cdots \to o \to}_{ar(f)} o$. Further we assume that each symbol $f \in \Sigma$ is assigned a finite set $\mathsf{Dir}(f) = \{1, \ldots, ar(f)\}$ of *directions*, and we define $\mathsf{Dir}(\Sigma) = \bigcup_{f\in\Sigma} \mathsf{Dir}(f)$. Let D be a set of directions; a ***D*-tree** is just a prefix-closed subset of D^*, the free monoid of D. A Σ-*labelled ranked and ordered tree* (or simply a **Σ-labelled tree**) is a function $t : \mathsf{Dom}(t) \longrightarrow \Sigma$ such that $\mathsf{Dom}(t)$ is a $\mathsf{Dir}(\Sigma)$-tree, and for every node $\alpha \in \mathsf{Dom}(t)$, the Σ-symbol $t(\alpha)$ has arity k if and only if α has exactly k children and the set of its children is $\{\alpha 1, \ldots, \alpha k\}$. We write $\mathcal{T}^\infty(\Sigma)$ for the set of (finite and infinite) Σ-labelled trees.

Let Ξ be a set of typed symbols. Let $f \in \Xi$ and A be a type, we write $f : A$ to mean that f has type A. The set of ***(applicative) terms of type A generated from Ξ***, written $\mathcal{T}_A(\Xi)$, is defined by induction over the following rules. If $f : A$ is an element of Ξ then $f \in \mathcal{T}_A(\Xi)$; if $s \in \mathcal{T}_{A\to B}(\Xi)$ and $t \in \mathcal{T}_A(\Xi)$ then $s\ t \in \mathcal{T}_B(\Xi)$. For simplicity we write $\mathcal{T}(\Xi)$ to mean $\mathcal{T}_o(\Xi)$, the set of terms of ground type. Let t be a term, we write $t : A$ to mean that t is an term of type A. In case Ξ is a ranked alphabet (and so every Ξ-symbol has an order-0 or order-1 type as determined by its arity) we identify terms in $\mathcal{T}(\Xi)$ with the finite trees in $\mathcal{T}^\infty(\Xi)$.

2.2 Recursion schemes

For each type A, we assume an infinite set Var_A of variables of type A, such that Var_A and Var_B are disjoint whenever $A \neq B$; and we write Var for the union of Var_A as A ranges over types. We use letters $x, y, \varphi, \psi, \chi, \xi$ etc. to range over variables.

A (deterministic) ***recursion scheme*** is a quadruple $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where

- Σ is a ranked alphabet of ***terminals*** (including a distinguished symbol $\bot : o$)
- $\mathcal{N}$ is a finite set of typed ***non-terminals***; we use upper-case letters F, H, etc. to range over non-terminals
- $S \in \mathcal{N}$ is a distinguished ***start symbol*** of type o
- $\mathcal{R}$ is a finite set of ***rewrite rules***, *one for each* non-terminal $F : (A_1, \cdots, A_n, o)$, of the form
$$F\,\xi_1\,\cdots\,\xi_n \;\to\; e$$
where each ξ_i is a variable of type A_i, and e is a term in $\mathcal{T}(\Sigma \cup \mathcal{N} \cup \{\xi_1, \cdots, \xi_n\})$. Note that the expressions on either side of the arrow are terms of ground type.

The ***order*** of a recursion scheme is defined to be the highest order of (the types of) its non-terminals.

In this paper we use recursion schemes as generators of Σ-labelled trees. Informally the *value tree*[2] $[\![G]\!]$ of (or the tree *generated* by) a recursion scheme G is a possibly infinite term

[2]We refer to the Σ-labelled tree generated by a recursion scheme as its *value* tree, because the name is a

(of ground type), *constructed from the terminals in* Σ, that is obtained, starting from the start symbol S, by unfolding the rewrite rules of G *ad infinitum*, replacing formal by actual parameters each time.

To define $[\![G]\!]$, we first introduce a map $(\cdot)^\perp : \bigcup_A \mathcal{T}_A(\Sigma \cup \mathcal{N}) \longrightarrow \bigcup_{A:ord(A)\leq 1} \mathcal{T}_A(\Sigma)$ that takes a term and replaces each non-terminal, together with its arguments, by $\perp$. We define $(\cdot)^\perp$ by structural recursion as follows: we let f range over Σ-symbols, and F over non-terminals in $\mathcal{N}$

$$\begin{array}{rcl} f^\perp & = & f \\ F^\perp & = & \perp \\ (st)^\perp & = & \begin{cases} \perp & \text{if } s^\perp = \perp \\ (s^\perp t^\perp) & \text{otherwise.} \end{cases} \end{array}$$

Clearly if $s \in \mathcal{T}(\Sigma \cup \mathcal{N})$ is of ground type, so is $s^\perp \in \mathcal{T}(\Sigma)$.

Next we define a one-step reduction relation $\to_G$ which is a binary relation over terms in $\mathcal{T}(\Sigma \cup \mathcal{N})$. Informally, $s \to_G s'$ just if s' is obtained from s by replacing some occurrence of a non-terminal F by the right-hand side of its rewrite rule in which all formal parameters are in turn replaced by their respective actual parameters, subject to the proviso that the F must occur at the head of a subterm of ground type. Formally $\to_G$ is defined by induction over the following rules:

- (*Substitution*). $Ft_1 \cdots t_n \to_G e[t_1/\xi_1, \cdots, t_n/\xi_n]$ where $F\xi_1 \cdots \xi_n \to e$ is a rewrite rule of G.
- (*Context*). If $t \to_G t'$ then $(st) \to_G (st')$ and $(ts) \to_G (t's)$.

Note that $\mathcal{T}^\infty(\Sigma)$ is a complete partial order with respect to the approximation ordering $\sqsubseteq$ defined by: $t \sqsubseteq t'$ just if $\mathsf{Dom}(t) \subseteq \mathsf{Dom}(t')$ and for all $w \in \mathsf{Dom}(t)$, we have $t(w) = \perp$ or $t(w) = t'(w)$. *I.e.* t' is obtained from t by replacing some $\perp$-labelled nodes by Σ-labelled trees. If one views G as a rewrite system, it is a consequence of the Church-Rosser property [18] that the set $\{\, t^\perp \in \mathcal{T}^\infty(\Sigma) : \text{there is a finite reduction sequence } S = t_0 \to_G \cdots \to_G t_n = t \,\}$ is directed. Hence, we can finally define the Σ-labelled ranked tree $[\![G]\!]$, called the ***value tree*** of (or the tree ***generated*** by) G:

$$[\![G]\!] = \sup\{\, t^\perp \in \mathcal{T}^\infty(\Sigma) : \text{there is a finite reduction sequence } S = t_0 \to_G \cdots \to_G t_n = t \,\}.$$

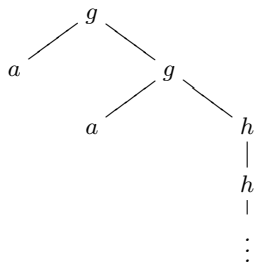
We write $\boldsymbol{RecTree_n\Sigma}$ for the class of value trees $[\![G]\!]$ where G ranges over order-n recursion schemes.

Exampl 1. *Let G_1 be the order-2 recursion scheme with non-terminals $\{S : o, H : (o,o), F : ((o,o),o)\}$, variables $\{z : o, \varphi : (o,o)\}$, terminals g, h, a of arity $2, 1, 0$ respectively, and the following rewrite rules:*

$$\begin{array}{rcl} S & \to & H\, a \\ H\, z & \to & F\, (g\, z) \\ F\, \varphi & \to & \varphi\, (\varphi\, (F\, h)) \end{array}$$

good counterpoint to *computation tree*. We have in mind here the distinction between *value* and *computation* emphasized by Moggi [50]. The idea is that the value tree is obtained from the computation tree by a (possibly infinite) process of evaluation.

The value tree $[\![G]\!]$ *is the* Σ*-labelled tree representing the infinite term* $g\,a\,(g\,a\,(h\,(h\,(h\,\cdots))))$*:*

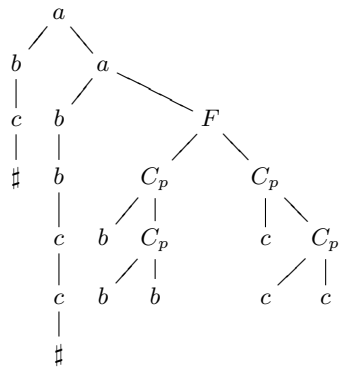


The only infinite path *in the tree is the node-sequence* $\varepsilon \cdot 2 \cdot 22 \cdot 221 \cdot 2211 \cdots$.

Exampl 2. *Let* G_2 *be the order-2 recursion scheme with non-terminals* $\{S : o,\ F : ((o,o),(o,o),o),\ C_p : ((o,o),(o,o),o,o)\}$*, variables* $\{x : o,\ \varphi : (o,o),\ \psi : (o,o)\}$*, terminals* $a, b, c, \sharp$ *of arity* $2, 1, 1, 0$ *respectively, and the following rewrite rules:*

$$\begin{array}{rcl} S & \to & F\,b\,c \\ F\,\varphi\,\psi & \to & a\,(\varphi\,(\psi\,\sharp))\,(F\,(C_p\,b\,\varphi)\,(C_p\,c\,\psi)) \\ C_p\,\varphi\,\psi\,x & \to & \varphi\,(\psi\,x) \end{array}$$

After some applications of the rules, one gets the following term:



The value tree $[\![G]\!]$ *is the* Σ*-labelled tree representing the infinite term*

$$a\,(b\,c\,\sharp)\,(a\,(b\,b\,c\,c\,\sharp)\,(a\,(b\,b\,b\,c\,c\,c\,\sharp)\,\cdots))$$

In particular, the path language of t *(i.e. the set of words obtained by considering the labels along a maximal branch) is* $\{a^\omega\} \cup \{a^k b^k c^k \sharp \mid k \geq 1\}$.

2.3 The safety constraint

The *safety* constraint on applicative terms may be regarded as a reformulation of Damm's *derived types* [25]. To define safety, we first introduce ***homogeneous types***. The type $(A_1, \cdots, A_n, o)$ is *homogeneous* just if each A_i is homogeneous, and $ord(A_1) \geq ord(A_2) \geq \cdots \geq ord(A_n)$. It follows that the ground type o and all order-1 types are homogeneous. In the following definition, suppose a term s has type A, then we write $ord(s) = ord(A)$.

Definition 1. *A rewrite rule $F\ x_1 \dots x_n \to t$ is **safe** just if*

(i) the type of F and of all subterms of t are homogeneous, and

(ii) for each subterm s of t that occurs in the operand position of an application, and for each $1 \leq i \leq n$, if x_i occurs in s then $ord(s) \leq ord(x_i)$.

We say that a recursion scheme is safe *just if all its rewrite rules are safe.*

It follows from the definition that all recursion schemes of order at most 1 are safe. For a study of safety in the setting of the simply-typed lambda calculus, see [7].

Exampl 3. *The scheme G_1 defined in Example 1 is unsafe because of the second rule. The subterm $g\,z$ occurs at an operand position and has order 1, but z has order 0.*

3 Collapsible pushdown automata (CPDA)

We introduce *(higher-order) collapsible pushdown automata (CPDA)*. An order-n CPDA, or n-CPDA for short, is just an order-n pushdown automaton (n-PDA), in the sense of [40], in which every non-$\perp$ symbol in the order-n stack has a *link* to a (necessarily lower-ordered) stack situated below it. In the following section we give an exposition where links are treated informally. A more formal treatment of the links is given in Section 3.2.

3.1 Stacks with links

Fix a stack alphabet Γ and a distinguished ***bottom-of-stack symbol*** $\perp \in \Gamma$. An ***order-0 stack*** (or simply ***0-stack***) is just a stack symbol. An ***order-$(n+1)$ stack*** (or simply ***$(n+1)$-stack***) s is a non-null sequence (written $[s_1 \cdots s_l]$) of n-stacks such that every non-$\perp$ Γ-symbol γ that occurs in s has a link to a stack of some order e (say, where $0 \leq e \leq n$) situated below it in s; we call the link an ***$(e+1)$-link***. The ***order*** of a stack s is written $ord(s)$.

As usual, the bottom-of-stack symbol $\perp$ cannot be popped from or pushed onto a stack. Thus we require an ***order-1 stack*** to be a non-null sequence $[\gamma_1 \cdots \gamma_l]$ of elements of Γ such that for all $1 \leq i \leq l$, $\gamma_i = \perp$ iff $i = 1$. We define $\perp_k$, the ***empty k-stack***, as follows: $\perp_0 = \perp$ and $\perp_{k+1} = [\perp_k]$.

We first define the operations pop_i and top_i with $i \geq 1$: $top_i(s)$ returns the top $(i-1)$-stack of s, and $pop_i(s)$ returns s with its top $(i-1)$-stack removed. Precisely let $s = [s_1 \cdots s_{l+1}]$ be a stack with $1 \leq i \leq ord(s)$:

$$top_i(\underbrace{[s_1 \cdots s_{l+1}]}_{s}) = \begin{cases} s_{l+1} & \text{if } i = ord(s) \\ top_i(s_{l+1}) & \text{if } i < ord(s) \end{cases}$$

$$pop_i(\underbrace{[s_1 \cdots s_{l+1}]}_{s}) = \begin{cases} [s_1 \cdots s_l] & \text{if } i = ord(s) \text{ and } l \geq 1 \\ [s_1 \cdots s_l\ pop_i(s_{l+1})] & \text{if } i < ord(s) \end{cases}$$

By abuse of notation, we set $top_{ord(s)+1}(s) = s$. Note that $pop_i(s)$ is undefined if $top_{i+1}(s)$ is a one-element i-stack. For example $pop_2([[\perp\,\alpha\,\beta]])$ and $pop_1([[\perp\,\alpha\,\beta]\,[\perp]])$ are both undefined.

There are two kinds of *push* operations. We start with the *order*-1 *push*. Let γ be a non-$\perp$ stack symbol and $1 \leq e \leq ord(s)$, we define a new stack operation $push_1^{\gamma,e}$ that, when applied to s, first attaches a link from γ to the $(e-1)$-stack *immediately* below the top $(e-1)$-stack

of s, then pushes γ (with its link) onto the top 1-stack of s. Formally for $1 \leq e \leq ord(s)$ and $\gamma \in (\Gamma \setminus \{\perp\})$, we define

$$push_1^{\gamma,e}(\underbrace{[s_1 \cdots s_{l+1}]}_{s}) = \begin{cases} [s_1 \cdots s_l \, push_1^{\gamma,e}(s_{l+1})] & \text{if } e < ord(s) \\ [s_1 \cdots s_l \, s_{l+1} \, \gamma^\dagger] & \text{if } e = ord(s) = 1 \\ [s_1 \cdots s_l \, push_1^{\widehat{\gamma}}(s_{l+1})] & \text{if } e = ord(s) \geq 2 \text{ and } l \geq 1 \end{cases}$$

where

- $\gamma^\dagger$ denotes the symbol γ with a link to the 0-stack s_{l+1}
- $\widehat{\gamma}$ denotes the symbol γ with a link to the $(e-1)$-stack s_l; and we define

$$push_1^{\widehat{\gamma}}(\underbrace{[t_1 \cdots t_{r+1}]}_{t}) = \begin{cases} [t_1 \cdots t_r \, push_1^{\widehat{\gamma}}(t_{r+1})] & \text{if } ord(t) > 1 \\ [t_1 \cdots t_{r+1} \, \widehat{\gamma}] & \text{otherwise } \textit{i.e. } ord(t) = 1 \end{cases}$$

The higher-order $push_j$, where $j \geq 2$, simply duplicates the top $(j-1)$-stack of s, including all the links. Precisely, let $s = [s_1 \cdots s_{l+1}]$ be a stack with $2 \leq j \leq ord(s)$:

$$push_j(\underbrace{[s_1 \cdots s_{l+1}]}_{s}) = \begin{cases} [s_1 \cdots s_{l+1} \, s_{l+1}] & \text{if } j = ord(s) \\ [s_1 \cdots s_l \, push_j(s_{l+1})] & \text{if } j < ord(s) \end{cases}$$

Note that in case $j = ord(s)$ above, the link structure of s_{l+1} is preserved by the copy that is pushed on top by $push_j$.

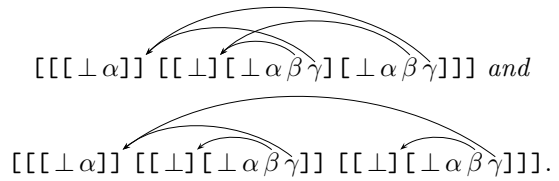
Finally there is an important operation called *collapse*. We say that the n-stack s_0 is a ***prefix*** of an n-stack s, written $s_0 \leq s$, just in case s_0 can be obtained from s by a sequence of (possibly higher-order) *pop* operations. Take an n-stack s where $s_0 \leq s$, for some n-stack s_0, and $top_1\, s$ has a link to $top_e(s_0)$. Then *collapse* s is defined to be s_0.

Exampl 4. *When displaying n-stacks in examples, we use bent arrows to denote links; however to avoir clutter we shall omit 1-links (indeed by construction they can only point to the symbol directly below), writing e.g.* `[[⊥][⊥α β]]` *instead of* `[[⊥][⊥` $\overset{\curvearrowleft}{\alpha}$ $\overset{\curvearrowleft}{\beta}$`]]`.

Take the 3-stack $s =$ `[[[⊥α]] [[⊥][⊥α]]]`. *We have*

$$\begin{aligned} push_1^{\beta,2}(s) &= [[[\perp\alpha]] \; [[\perp] \overset{\curvearrowleft}{[\perp\alpha\,\beta]}]] \\ collapse\,(push_1^{\beta,2}(s)) &= [[[\perp\alpha]] \; [[\perp]]] \\ \underbrace{push_1^{\gamma,3}(push_1^{\beta,2}(s))}_{\theta} &= \overset{\curvearrowleft}{[[[\perp\alpha]]} \; \overset{\curvearrowleft}{[[\perp] \overset{\curvearrowleft}{[\perp\alpha\,\beta\,\gamma]}]]}. \end{aligned}$$

Then $push_2(\theta)$ *and* $push_3(\theta)$ *are respectively*



$[[[\perp\alpha]]\ [[\perp][\perp\alpha\beta\gamma][\perp\alpha\beta\gamma]]]$ *and*

$[[[\perp\alpha]]\ [[\perp][\perp\alpha\beta\gamma]]\ [[\perp][\perp\alpha\beta\gamma]]]$.

We have $collapse\,(push_2(\theta)) = collapse\,(push_3(\theta)) = collapse(\theta) = [[[\perp\alpha]]]$.

3.2 A formal definition of CPDA stack operations

One way to give a formal semantics of the stack operations is to work with appropriate numeric representations of the links. In [41], it has been shown how this can be done in the order-2 case in the setting of panic automata. Here we use a different encoding of stacks with links that works for all orders. The presentation follows Kartzow [37].

The idea is simple: take an order-n stack s and suppose that there is a link from (a particular occurrence of) a symbol γ in s to some $(e-1)$-stack s', and that s' is the k-th element of the e-stack that contains it. In the formal definition, a ***symbol-with-link*** of an order-n CPDA is written $\gamma^{(e,k)}$, where $\gamma \in \Gamma$, $1 \le e \le n$ and $k \ge 1$, Purely for convenience, we require that if $\gamma = \perp$ then $e = 1$ and $k = 0$.

The set Op_n of order-n CPDA ***stack operations*** comprises four types of operations:

1. pop_k for each $1 \le k \le n$
2. $push_j$ for each $2 \le j \le n$
3. $push_1^{\gamma,e}$ for each $1 \le e \le n$ and each $\gamma \in (\Gamma \setminus \{\perp\})$, and
4. *collapse*.

We begin by defining an operation that truncates a stack.

$$bot_i^k(\underbrace{[t_1 \cdots t_{r+1}]}_{t}) = \begin{cases} [t_1 \cdots t_k] & \text{if } ord(t) = i \text{ and } k \le r \\ [t_1 \cdots t_r\ bot_i^k(t_{r+1})] & \text{if } ord(t) < i \text{ and } k \le r \end{cases}$$

We can now define our stack operations. Let $1 \le e \le ord(s)$. We first define $push_1^{\gamma,e}$

We first define $push_1^{\gamma}$ to aid in the definition of $push_1^{\gamma,e}$.

$$push_1^{\gamma}(\underbrace{[t_1 \cdots t_{r+1}]}_{t}) = \begin{cases} [t_1 \cdots t_r\ push_1^{\gamma}(t_{r+1})] & \text{if } ord(t) > 1 \\ [t_1 \cdots t_{r+1}\,\gamma] & \text{otherwise } \textit{i.e. } ord(t) = 1 \end{cases}$$

Then we have

$$push_1^{\gamma,e}(t) = push_1^{\gamma^{(e,k)}}$$

assuming $top_{e+1}(t) = [s_1 \cdots s_{k+1}]$ if $e > 1$, and $top_2(t) = [s_1 \cdots s_k]$ for $e = 1$. We are now ready to define the *collapse* operation by letting

$$collapse(s) \quad = \quad bot_e^k(s) \quad \text{where } top_1(s) = \gamma^{(e,k)} \text{ and } k > 0$$

One can think of the *collapse* operation as a generalisation of the pop_k operation for any $k > 1$ as we have for any stack s and any $k > 1$ that $pop_k(s) = collapse(push_1^{\gamma,k}(s))$ for an arbitrary dummy symbol γ.

Now for $2 \leq j \leq ord(s)$:

$$push_j(\underbrace{[s_1 \cdots s_{l+1}]}_{s}) = \begin{cases} [s_1 \cdots s_{l+1}\ s_{l+1}] & \text{if } j = ord(s) \\ [s_1 \cdots s_l\ push_j(s_{l+1})] & \text{if } j < ord(s)\ . \end{cases}$$

Note that, as an easy consequence of the definitions of the $push_1^{\gamma,e}$ and the $push_k$ operations, a link of order e always points to a $(e-1)$-stack inside the current e-stack.

Exampl 5. *Let us now revisit Example 4. Take the 3-stack* $s = [[[\bot\,\alpha]]\ [[\bot]\,[\bot\,\alpha]]]$. *(To save writing, we omit the superscripts of the form* $(1,k)$*.) We have*

$$\begin{aligned} push_1^{\beta,2}(s) &= [[[\bot\,\alpha]]\ [[\bot]\,[\bot\,\alpha\,\beta^{(2,1)}]]] \\ push_1^{\gamma,3}(push_1^{\beta,2}(s)) &= [[[\bot\,\alpha]]\ [[\bot]\,[\bot\,\alpha\,\beta^{(2,1)}\,\gamma^{(3,1)}]]] \\ push_2(push_1^{\gamma,3}(push_1^{\beta,2}(s))) &= [[[\bot\,\alpha]]\ [[\bot]\,[\bot\,\alpha\,\beta^{(2,1)}\,\gamma^{(3,1)}]\,[\bot\,\alpha\,\beta^{(2,1)}\,\gamma^{(3,1)}]]] \\ push_3(push_1^{\gamma,3}(push_1^{\beta,2}(s))) &= [[[\bot\,\alpha]]\ [[\bot]\,[\bot\,\alpha\,\beta^{(2,1)}\,\gamma^{(3,1)}]]\ [[\bot]\,[\bot\,\alpha\,\beta^{(2,1)}\,\gamma^{(3,1)}]]] \end{aligned}$$

and we have

$$collapse(push_2(push_1^{\gamma,3}(push_1^{\beta,2}(s)))) = collapse(push_3(push_1^{\gamma,3}(push_1^{\beta,2}(s)))) = [[[\bot\,\alpha]]]$$

Note that in the sequel we will use the informal presentation of stacks with links rather than the formal one.

3.3 Tree-generating CPDA

Collapsible pushdown automata are a generalization (to all finite orders) of *pushdown automata with links* [2, 3], which are essentially the same as *panic automata* [41].

We define collapsible pushdown automata (CPDA) as automata with a finite control and a stack with links as memory.

Definition 2. *An* ***order-n (deterministic) collapsible pushdown automaton (n-CPDA)*** *is a 5-tuple* $\mathcal{A} = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_I \rangle$ *where* A *is an input alphabet and* ε *is a special symbol,* Γ *is a stack alphabet,* Q *is a finite set of control states,* $q_I \in Q$ *is the initial state, and* $\delta : Q \times \Gamma \times (A \cup \{\varepsilon\}) \to Q \times Op_n$ *is a transition (partial) function such that, for all* $q \in Q$ *and* $\gamma \in \Gamma$*, if* $\delta(q, \gamma, \varepsilon)$ *is defined then for all* $a \in A$*,* $\delta(q, \gamma, a)$ *is undefined i.e. if an* ε*-transition can be taken, then no other transitions are possible.*

As CPDA will be used to generate ranked tree (as explained below), A will always be here of the form $\{1, \ldots, d\}$ *for some integer* d*.*

In the special case where $\delta(q, \gamma, \varepsilon)$ *is undefined for all* $q \in Q$ *and* $\gamma \in \Gamma$ *we refer to* $\mathcal{A}$ *as an* ε*-free n-CPDA.*

Configurations *of an n-CPDA are pairs of the form* (q, s) *where* $q \in Q$ *and* s *is an n-stack with links over* Γ*; we call* $(q_I, \bot_n)$ *the* ***initial configuration****.*

An n-CPDA $\mathcal{A} = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_I \rangle$ *naturally defines an* $(A \cup \{\varepsilon\})$*-labelled transition graph* $\mathrm{Graph}(\mathcal{A}) := (V, E \subseteq V \times (A \cup \{\varepsilon\}) \times V)$ *whose vertices* V *are the configurations of* $\mathcal{A}$ *and whose edge relation* E *is given by:* $((q,s),a,(q',s')) \in E$ *iff* $\delta(q, top_1(s), a) = (q', op)$ *and* $s' = op(s)$*. Such a graph is called an* ***n-CPDA graph****. We shall use the notation* $v \xrightarrow{a} v'$ *to mean that* $(v,a,v') \in E$*, and* $v \xrightarrow{a_1 a_2 \cdots a_\ell} v'$ *to mean that there exist* $v_0, \ldots, v_\ell \in V$ *such that* $v_0 = v$*,* $v_\ell = v'$ *and* $v_i \xrightarrow{a_{i+1}} v_{i+1}$ *for all* $0 \leq i < \ell$*.*

Note that one can transform $\mathcal{A}$, while preserving its transition graph, so that in every configuration (q,s) reachable from the initial one, whenever $\delta(q, top_1\, s, a) = (q', op)$ is defined, so is $op(s)$ *i.e.* whenever a transition is possible, the corresponding stack action is well-defined. Such a transformation can be obtained by storing in the stack extra information about feasibility of the pop_k operation[3]. In the following we always assume that we are in such a setting.

Exampl 6. *Consider the following* 2*-CPDA (that actually does not make use of links)* $\mathcal{A} = \langle \{1,2,\varepsilon\}, \{\bot, \alpha\}, \{q_a, q_b, q_c, q_\sharp, \widetilde{q}_a, \widetilde{q}_b, \widetilde{q}_c\}, \delta, \widetilde{q}_a \rangle$ *with* δ *as follows (we only give those transitions that may happen):*

- $\delta(\widetilde{q}_a, \bot, \varepsilon) = \delta(q_a, \alpha, 2) = (q_a, push_1^\alpha)$*;*
- $\delta(q_a, \alpha, 1) = (\widetilde{q}_b, push_2)$*;*
- $\delta(\widetilde{q}_b, \alpha, \varepsilon) = \delta(q_b, \alpha, 1) = (q_b, pop_1)$*;*
- $\delta(q_b, \bot, 1) = (\widetilde{q}_c, pop_2)$*;*
- $\delta(\widetilde{q}_c, \alpha, \varepsilon) = \delta(q_c, \alpha, 1) = (q_c, pop_1)$*;*
- $\delta(q_c, \bot, 1) = (q_\sharp, id)$ *where* id *is the operation that leaves the stack unchanged;*
- $\delta(q_\sharp, \bot, \_)$ *is undefined.*

Then $\mathrm{Graph}(\mathcal{A})$ *is given in Figure 1.*

We now explain how to define from $\mathcal{A}$ a $(\Sigma \cup \{\bot\})$-labelled ranked tree t for a ranked alphabet Σ where $\bot$ is an additional symbol of arity 0. The idea is first to unfold $\mathrm{Graph}(\mathcal{A})$, then to contract the ε-transitions, and finally to label the nodes carefully.

A vertex v in $\mathrm{Graph}(\mathcal{A})$ is ***non-productive*** if it is the source of an infinite path labelled by ε^ω *i.e.* for every $k \geq 0$ there exists v_k such that $v \xrightarrow{\varepsilon} v_1 \xrightarrow{\varepsilon} v_2 \xrightarrow{\varepsilon} v_3 \cdots$ Otherwise v is said to be ***productive***.

First we assume that $A = \{1, \cdots, d\}$ for some $d \geq 1$, and whenever $\{a \in A \mid (q,\gamma,a) \in \mathsf{Dom}(\delta)\}$ has k elements then it is $\{1, \cdots, k\}$. And we consider a partial function $\rho : Q \times \Gamma \rightharpoonup \Sigma$ such that for every q and γ if $(q,\gamma,\varepsilon) \notin \mathsf{Dom}(\delta)$ then $(q,\gamma) \in \mathsf{Dom}(\rho)$ and $\{a \in A \mid (q,\gamma,a) \in \mathsf{Dom}(\delta)\} = \mathsf{Dir}(\rho(q,\gamma))$; We will use the function ρ to define the node labels of the tree t being constructed.

We set $\mathsf{Dom}(t)$ to be the prefix-closed subset of A^* defined by

$$\mathsf{Dom}(t) := \{w \in A^* \mid \exists v \in V \,.\, (q_I, \bot_n) \xrightarrow{w} v\}.$$

Thanks to determinism, for all $w \in \mathsf{Dom}(t)$ there is a unique vertex v_w such that $(q_I, \bot_n) \xrightarrow{w} v_w$ and such that $v_w \xrightarrow{\varepsilon} v$ holds whenever $(q_I, \bot_n) \xrightarrow{w} v$.

[3]This can be done by extending higher-order PDA to allow the annotation of each order-k stack with the feasibility of the pop_k operation, which can in turn be transformed into a standard higher-order PDA following the remark on Page 9 of Knapik *et al.* [40].

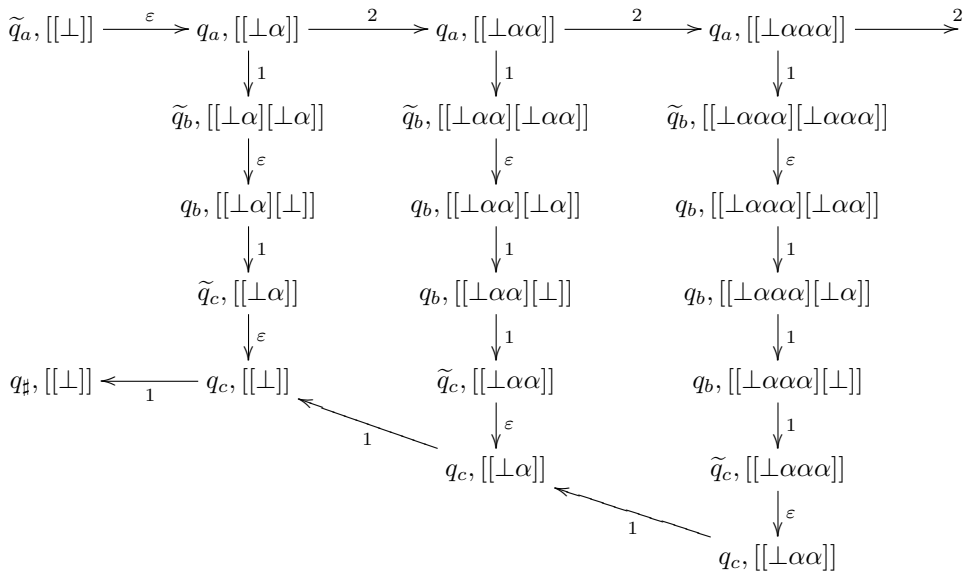


Figure 1: Transition graph of the CPDA of Example 6

In case v_w is productive, define (q_w, s_w) to be the unique configuration with $v_w \xrightarrow{\varepsilon} (q_w, s_w)$ that is not the source of an ε transition, *i.e.* that is such that $(q_w, top_1(s_w), \varepsilon) \notin \mathsf{dom}(\delta)$.

We can finally define

$$t(w) := \begin{cases} \bot & \text{if } v_w \text{ is non-productive;} \\ \rho(q_w, top_1(s_w)) & \text{otherwise.} \end{cases}$$

Hence there are two kinds of leaves in t: those labelled by symbols in Σ which correspond to dead-ends in Graph($\mathcal{A}$), and those labelled by $\bot$ which correspond to non-productive vertices in Graph($\mathcal{A}$). Note the analogy with trees generated by recursion schemes, where $\bot$ is used to label those leaves that correspond to an infinite sequence of "non-productive" rewritings.

Exampl 7. *Consider the CPDA $\mathcal{A}$ from Example 6 and let $\rho(q_a, \_) = a$, $\rho(q_b, \_) = b$, $\rho(q_c, \_) = c$, $\rho(q_\sharp, \_) = \sharp$, where $\_$ stands for any stack symbol. Then, the tree generated by $\mathcal{A}$ and ρ is the same as the one generated by the order-2 recursion scheme of Example 2.*

Remark 1. *Thanks to ε-transitions, we can safely assume that the labelling function ρ only depends on the control state i.e. $\rho : Q \to \Sigma$ instead of $\rho : Q \times \Gamma \to \Sigma$, as in Example 7. One can always encode the current top stack symbol in the control state: after each transition, perform an ε-transition that updates the control state according to the top stack symbol.*

Remark 2. *A natural variant of CPDA allows the execution of several stack operations per transition i.e. by defining $\delta : Q \times \Gamma \times (A \cup \{\varepsilon\}) \to Q \times Op_n^*$. To simulate such a variant by a standard CPDA, it suffices to add intermediate states to track a sequence of stack operations by a finite sequence of ε-transitions.*

Remark 3. *By allowing several stack operations per transition, one can get rid of the states by encoding them in the stack symbols. In this setting, given a CPDA without state (i.e. with*

a dummy single state) but allowing several stack operations per transition, a ranked tree can be generated by unfolding the transition graph and taking the ε-closure (i.e. we contract each ε-labelled edge, merging its source and target vertices). The nodes are labelled according to a function $\rho : \Gamma \to \Sigma$. It is easy to check that such a variant CPDA is equi-expressive with the standard CPDA for generating trees.

Remark 4. *In [40, 34], deterministic higher-order pushdown automata and CPDA are used directly as tree-accepting device in a top-down fashion, allowing silent moves. When reading a node of an input tree in a given state, the automaton may make a number of ε-transitions (hence changing both the stack and the state) and then it branches by sending a copy of the automaton to read each child node in a state prescribed by the transition function. Thanks to the determinism,* exactly one *tree is accepted by the automaton. It is easy to see this definition coincides with our notion of tree generation by an n-CPDA. Essentially branching corresponds to unfolding, and ε-transitions to taking the ε-closure of the unfolding.*

4 The Equi-Expressivity Theorem

In this paper we prove the following theorem.

Theorem 1 (Equi-Expressivity)**.** *Order-n recursion schemes and n-CPDA are equi-expressive for generating trees. I.e. we have the following.*

(i) Let G be an order-n recursion scheme over Σ and let t be its value tree. There is an order-n CPDA $\mathcal{A} = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_0 \rangle$ and a function $\rho : Q \to \Sigma$ such that t is the tree generated by $\mathcal{A}$ and ρ.

(ii) Let $\mathcal{A} = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_0 \rangle$ be an order-n CPDA, and let t be the Σ-labelled tree generated by $\mathcal{A}$ and a function $\rho : Q \to \Sigma$. There is an order-n recursion scheme over Σ whose value tree is t.

Further the inter-translations between schemes and CPDA are polytime computable.

Theorem 1 extends to all recursion schemes the following result [39] about *safe* recursion schemes. An n-PDA is just an n-CPDA that never performs a *collapse*.

Theorem 2 ([39])**.** *Order-n safe recursion schemes and n-PDA are equi-expressive for generating trees. Moreover the inter-translations between safe schemes and n-PDA are polytime computable.*

Theorem 1 also extends to all finite orders a similar result from [41] restricted to order 2.

The rest of this paper is devoted to the proof of Theorem 1: Section 5 proves that schemes are at least as expressive as CPDA (Theorem 3) and Section 6 proves that CPDA are at least as expressive as schemes (Theorem 6).

5 From CPDA to recursion schemes

For the rest of this section we fix an order-n CPDA $\mathcal{A} = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_1 \rangle$ where $Q = \{q_1, \cdots, q_m\}$ and $m \geq 1$. We shall first introduce a representation of stacks and configurations of $\mathcal{A}$ by terms which are then organised into a recursion scheme. Finally we show that the labelled transition system associated with the recursion scheme is identical to the labelled transition graph of $\mathcal{A}$.

5.1 Term representation of stacks and configurations

We start by defining, for every $0 \leq k \leq n$ a type denoted $\mathbf{k}$ that will later be used to type the behaviour of a k-stack. First we identify the ground type o with a new type denoted $\mathbf{n}$. Inductively, for each $0 \leq k < n$ we define a type

$$\mathbf{k} \quad = \quad (\mathbf{k+1})^m \to (\mathbf{k+1})$$

where, for types A and B, we write $A^m \to B$ as a shorthand for $\underbrace{A \to \cdots \to A}_{m} \to B$. In particular, for every $0 \leq k < n$, we have

$$\mathbf{k} \;=\; (\mathbf{k+1})^m \to (\mathbf{k+2})^m \to \cdots \to \mathbf{n}^m \to \mathbf{n}$$

We also introduce a non-terminal Void_k of type $\mathbf{k}$ for each $0 \leq k \leq n$.

Assume s is an order-n stack and p is a control state of $\mathcal{A}$. In the sequel, we will define, for every $0 \leq k \leq n$, a term $[\![s]\!]^p_k : \mathbf{k}$ that represents the *behaviour* of the topmost k-stack in s, *i.e.* $top_{k+1}(s)$. To understand why $[\![s]\!]^p_k$ is of type $\mathbf{k}$ one can view an order-k stack as acting on order-$(k+1)$ stacks: for every order-$(k+1)$ stack we can build a new order-$(k+1)$ stack by pushing an order-k stack on top of it. This behaviour has the type $(\mathbf{k+1}) \to (\mathbf{k+1})$. However, for technical reasons, when dealing with control states and configurations, we need to work with m copies of each stack, one for each control state. Hence we view a k-stack as mapping m copies of an order-$(k+1)$ stack to a single order-$(k+1)$ stack. This explains why $\mathbf{k}$ is defined to be $(\mathbf{k+1})^m \to (\mathbf{k+1})$.

For every stack symbol γ, every $1 \leq e \leq n$ and every state $p \in Q$, we introduce a non-terminal

$$\mathcal{F}^{\gamma,e}_p \;:\; \mathbf{e}^m \to \mathbf{1}^m \to \cdots \to \mathbf{n}^m \to \mathbf{n}$$

Note that the type of $\mathcal{F}^{\gamma,e}_p$ is non-homogeneous.

For every $0 \leq k \leq n$, every state p and every order-n stack s whose topmost stack symbol is $top_1(s) = \gamma$ with an $(e+1)$-link, we inductively define the following term of order $\mathbf{k} = (\mathbf{k+1})^m \to \cdots \to \mathbf{n}^m \to \mathbf{n}$

$$[\![s]\!]^p_k = \mathcal{F}^{\gamma,e}_p [\![collapse(s)]\!]^{\overline{q}}_e [\![pop_1(s)]\!]^{\overline{q}}_1 [\![pop_2(s)]\!]^{\overline{q}}_2 \cdots [\![pop_k(s)]\!]^{\overline{q}}_k$$

where $[\![t]\!]^{\overline{q}}_h$ is a shorthand for the sequence $[\![t]\!]^{q_1}_h [\![t]\!]^{q_2}_h \cdots [\![t]\!]^{q_m}_h$, and if $pop_i(s)$ is undefined then we adopt the convention that $[\![pop_i(s)]\!]^{\overline{q}}_i$ means $\underbrace{\mathrm{Void}_i \cdots \mathrm{Void}_i}_{m}$.

The preceding definition is well-founded: every stack in the definition of $[\![s]\!]^p_k$ has fewer symbols than s. Intuitively $[\![s]\!]^p_k$ represents the top k-stack of the configuration (p, s).

Let s and t be order-n stacks with links and let $1 \leq k \leq n$. We define s and t are ***top_k-identical*** as follows (where $pop^j_k(s)$ denotes the stack obtained from s by k successive applications of the pop_k function):

- s and t are top_1-*identical* just if $top_1(s) = top_1(t)$, and $collapse(s)$ and $collapse(t)$ are top_{e+1}-identical where $top_1(s)$ has an $(e+1)$-link
- for $k > 1$, s and t are top_k-*identical* just if for every $j \geq 0$, $pop^j_{k-1}(s)$ is defined if and only if $pop^j_{k-1}(t)$ is defined, and if so, $pop^j_{k-1}(s)$ and $pop^j_{k-1}(t)$ are top_{k-1}-identical.

Taking $j = 0$ in the second item, we note that if s and t are top_k identical then they are also $top_{k'}$-identical for any $1 \leq k' \leq k$. The preceding definition is well-founded because it always refers to stacks with fewer symbols than s or t.

Lemma 1. *Let s and t be order-n stacks with links, and let $0 \leq k \leq n-1$. If s and t are top_{k+1}-identical then $[\![s]\!]^p_k = [\![t]\!]^p_k$ for every state p.*

Proof. The proof is by induction on the maximum of the respective sizes of s and t, and once that is fixed we reason by induction on k.

The base case of s and t containing only the bottom-of-stack symbol is trivial. Assume that the property holds for every pair of stacks, each with no more than N symbols for some $N > 0$, and consider stacks s and t, the larger of the two has size $N+1$. Assume that s and t are top_{k+1}-identical for some $k \geq 0$. We now reason by induction on k.

Suppose s and t are top_1-identical. By definition, we have that $top_1(s) = top_1(t) = (\gamma, e)$ where $\gamma \in \Gamma$ and $1 \leq e \leq n$, and that $collapse(s)$ and $collapse(t)$ are top_{e+1}-identical. As $collapse(s)$ and $collapse(t)$ have size bounded by N, by the induction hypothesis, we have $[\![collapse(s)]\!]^{\overline{q}}_e = [\![collapse(t)]\!]^{\overline{q}}_e$. Thus it follows immediately that $[\![s]\!]^p_0 = [\![t]\!]^p_0$.

Now take $k \geq 0$ and assume that the property is established for each $h \leq k$. We consider the case of $k+1$. Assume that s and t are top_{k+2}-identical. It follows that for each $h \leq k$, if $pop_h(s)$ (equivalently $pop_h(t)$) is defined then $pop_h(s)$ and $pop_h(t)$ are top_{h+1}-identical, and so, by the induction hypothesis, $[\![pop_h(s)]\!]^q_h = [\![pop_h(t)]\!]^q_h$ for every state q. Because s and t are top_{k+2}-identical they are also top_1-identical and then by definition, we also have $top_1(s) = top_1(t) = (\gamma, e)$ for some $\gamma \in \Gamma$, and $collapse(s)$ and $collapse(t)$ are top_{e+1}-identical. As $collapse(s)$ and $collapse(t)$ have size bounded by N, by the induction hypothesis, we have $[\![collapse(s)]\!]^{\overline{q}}_e = [\![collapse(t)]\!]^{\overline{q}}_e$.

By definition

$$[\![s]\!]^p_{k+1} = \mathcal{F}^{\gamma,e}_p [\![collapse(s)]\!]^{\overline{q}}_e [\![pop_1(s)]\!]^{\overline{q}}_1 \cdots [\![pop_{k+1}(s)]\!]^{\overline{q}}_{k+1}$$

and

$$[\![t]\!]^p_{k+1} = \mathcal{F}^{\gamma,e}_p [\![collapse(t)]\!]^{\overline{q}}_e [\![pop_1(t)]\!]^{\overline{q}}_1 \cdots [\![pop_{k+1}(t)]\!]^{\overline{q}}_{k+1}$$

Now for any n-stack r define j_r be the maximal j such that $pop^j_{k+1}(r)$ is defined. In particular we have $j_s = j_t$. If $j_s = 0$, $[\![pop_{k+1}(s)]\!]^{\overline{q}}_{k+1} = [\![pop_{k+1}(t)]\!]^{\overline{q}}_1 = \mathrm{Void}_{k+1} \cdots \mathrm{Void}_{k+1}$, and so $[\![s]\!]^p_{k+1} = [\![t]\!]^p_{k+1}$. If $j_s > 0$, note that $j_{pop_{k+1}(s)} = j_{pop_{k+1}(t)} = j_s - 1$ and $pop_{k+1}(s)$ and $pop_{k+1}(t)$ are $top_{(k+2)}$-identical. Thus, inductively (on j_s), we have $[\![pop_{k+1}(s)]\!]^q_{k+1} = [\![pop_{k+1}(t)]\!]^q_{k+1}$ for every state q. Hence we conclude that $[\![s]\!]^p_{k+1} = [\![t]\!]^p_{k+1}$. □

5.2 Associated rewrite rules

With every pair $\theta = (q, op) \in Q \times Op_n$, we associate a rewrite rule

$$\mathcal{F}^{\gamma,e}_p \, \overline{\Phi} \, \overline{\Psi_1} \cdots \overline{\Psi_n} \quad \xrightarrow{\theta} \quad \Xi_\theta$$

where for each $0 \leq j \leq n$ we have $\overline{\Psi_j} = \Psi_{j,1} \cdots \Psi_{j,m}$ is a sequence of variables, with each $\Psi_{j,i} : \mathbf{j}$; similarly $\overline{\Phi} = \Phi_1 \cdots \Phi_m$ is a sequence of variables, with each $\Phi_i : \mathbf{e}$.

The shape of Ξ_θ depends on op, as shown in Table 1, where $\langle \mathcal{F}^{\gamma,e}_i \, \overline{\Phi} \, \overline{\Psi_1} \cdots \overline{\Psi_k} \,|\, i \rangle$ is a shorthand for the sequence

$$\mathcal{F}^{\gamma,e}_{q_1} \, \overline{\Phi} \, \overline{\Psi_1} \cdots \overline{\Psi_k} \quad \mathcal{F}^{\gamma,e}_{q_2} \, \overline{\Phi} \, \overline{\Psi_1} \cdots \overline{\Psi_k} \quad \cdots \quad \mathcal{F}^{\gamma,e}_{q_m} \, \overline{\Phi} \, \overline{\Psi_1} \cdots \overline{\Psi_k} : \mathbf{k^m}$$

The preceding labelled rewrite rules induce a θ-indexed family of *outermost* labelled one-step transition relations $\xrightarrow{\theta} \subseteq \mathcal{T}^0(\mathcal{N}_\mathcal{A}) \times \mathcal{T}^0(\mathcal{N}_\mathcal{A})$, where θ ranges over $Q \times Op_n$. Informally $M \xrightarrow{\theta}$

Table 1: Definition of Ξ_θ

Cases of op	**Corresponding** Ξ_θ **where** $\theta = (q, op)$
$push_1^{\gamma',e'}$	$\mathcal{F}_q^{\gamma',e'}\ \overline{\Psi_{e'}}\ \langle \mathcal{F}_i^{\gamma,e}\ \overline{\Phi}\ \overline{\Psi_1} \mid i\rangle\ \overline{\Psi_2}\ \cdots\ \overline{\Psi_n}$
$push_k$	$\mathcal{F}_q^{\gamma,e}\ \overline{\Phi}\ \overline{\Psi_1} \cdots \overline{\Psi_{(k-1)}} \langle \mathcal{F}_i^{\gamma,e}\ \overline{\Phi}\ \overline{\Psi_1} \cdots \overline{\Psi_k} \mid i\rangle \overline{\Psi_{(k+1)}} \cdots \overline{\Psi_n}$
pop_k	$\Psi_{k,q}\ \overline{\Psi_{k-1}} \cdots \overline{\Psi_n}$
$collapse$	$\Phi_q\ \overline{\Psi_{e-1}} \cdots \overline{\Psi_n}$

M' just if M' is obtained from M by replacing the *head* (equivalently, outermost) non-terminal by the right-hand side of the corresponding rewrite rule in which all formal parameters are in turn replaced by their respective actual parameters. Formally, for each $\theta = (q, op) \in Q \times Op_n$ and for each corresponding rewrite rule $\mathcal{F}_p^{\gamma,e}\overline{\Phi}\ \overline{\Psi_1} \cdots \overline{\Psi_n} \xrightarrow{\theta} \Xi_\theta$, we have the rule scheme

$$\mathcal{F}_p^{\gamma,e}\overline{L}\ \overline{M_1} \cdots \overline{M_n} \quad \xrightarrow{\theta} \quad \Xi_\theta[\overline{L}/\overline{\Phi}, \overline{M_1}/\overline{\Psi_1} \cdots, \overline{M_n}/\overline{\Psi_n}]$$

where $\overline{L}, \overline{M_1}, \cdots, \overline{M_n}$ range over sequences of terms that respect the type of $\mathcal{F}_p^{\gamma,e}$.

Note that each binary relation $\xrightarrow{\theta}$ is a partial function.

5.3 Correctness of the representation

Let (p, s) be a configuration of an order-n CPDA $\mathcal{A}$ and let $\theta = (q, op) \in Q \times Op_n$ be a transition. We say that (p, s) is ***θ-compatible*** just if θ is an applicable transition from (p, s) *i.e.* $\theta{=}\delta(p, top_1(s), a)$ for some $a \in (A \cup \{\varepsilon\})$ and $op(s)$ is defined. Recall that it is straightforward to transform $\mathcal{A}$ — without changing its expressivity — so that for every reachable configuration (p, s), if $\theta = (q, op){=}\delta(p, top_1(s), a)$ for some $a \in (A \cup \{\varepsilon\})$ then $op(s)$ is defined.

The following proposition relates the previous transition system with $\mathcal{A}$.

Proposition 1. *Let (p, s) be a configuration of $\mathcal{A}$ and $\theta = (q, op) \in Q \times Op_n$. If (p, s) is θ-compatible, then $[\![s]\!]_n^p \xrightarrow{\theta} t$ if and only if $t = [\![op(s)]\!]_n^q$.*

Proof. The proof is by a case analysis. Let $\theta = (q, op) \in Q \times Op_n$ and let (p, s) be θ-compatible. Set $C^{q_i} = [\![collapse(s)]\!]_e^{q_i} : \mathbf{e}$, and $T_k^{q_i} = [\![pop_k(s)]\!]_k^{q_i} : \mathbf{k}$ for every $1 \leq i \leq m$ and every $1 \leq k \leq n$. Then

$$[\![s]\!]_n^p = \mathcal{F}_p^{\gamma,e} \quad C^{q_1} \cdots C^{q_m} \quad T_1^{q_1} \cdots T_1^{q_m} \quad \cdots \quad T_n^{q_1} \cdots T_n^{q_m}$$

- Assume $op = push_1^{\gamma',e'}$. By definition we have

$$[\![push_1^{\gamma',e'}(s)]\!]_n^q = \mathcal{F}_q^{\gamma',e'} [\![collapse(push_1^{\gamma',e'}(s))]\!]_{e'}^{\overline{q}} \\ [\![pop_1(push_1^{\gamma',e'}(s))]\!]_1^{\overline{q}} \cdots [\![pop_n(push_1^{\gamma',e'}(s))]\!]_n^{\overline{q}}$$

We have $collapse(push_1^{\gamma',e'}(s)) = pop_{e'}(s)$, hence $[\![collapse(push_1^{\gamma',e'}(s))]\!]_{e'}^{\overline{q}} = T_{e'}^{\overline{q}}$. Further we have $pop_1(push_1^{\gamma',e'}(s)) = s$, hence $[\![pop_1(push_1^{\gamma',e'}(s))]\!]_1^{q_i} = \mathcal{F}_{q_i}^{\gamma,e}\ C^{\overline{q}}\ T_1^{\overline{q}}$. Finally, for each $j > 1$, we have $pop_j(push_1^{\gamma',e'}(s)) = pop_j(s)$, hence

$$[\![pop_j(push_1^{\gamma',e'}(s))]\!]_j^{\overline{q}} = T_j^{\overline{q}}.$$

Therefore, we have

$$[\![push_1^{\gamma',e'}(s)]\!]_n^q \quad = \quad \mathcal{F}_q^{\gamma',e'}\ \ T_{e'}^{\overline{q}}\ \ (\mathcal{F}_{q_1}^{\gamma,e}\ \ C^{\overline{q}}\ \ T_1^{\overline{q}})\ \ \cdots\ \ (\mathcal{F}_{q_m}^{\gamma,e}\ \ C^{\overline{q}}\ \ T_1^{\overline{q}})\ \ T_2^{\overline{q}}\ \ \cdots\ \ T_n^{\overline{q}}$$

On the other hand, it follows syntactically from the definition of $\xrightarrow{\theta}$ that the right-hand side of the preceding equation is the term t such that $[\![s]\!]_n^p \xrightarrow{\theta} t$. Hence one has $[\![s]\!]_n^p \xrightarrow{\theta} [\![push_1^{\gamma',e'}(s)]\!]_n^q$.

- Assume $op = push_k$. By definition we have

$$[\![push_k(s)]\!]_n^q = \mathcal{F}_q^{\gamma,e}\ [\![collapse(push_k(s))]\!]_e^{\overline{q}} [\![pop_1(push_k(s))]\!]_1^{\overline{q}}\ \cdots\ [\![pop_n(push_k(s))]\!]_n^{\overline{q}}$$

Note that we used the fact that the top_1 element in $push_k(s)$ is the same as that in s *i.e.* it is γ and has an $(e+1)$-link. Now if $e \leq k$, $collapse(push_k(s))$ and $collapse(s)$ are top_{e+1}-identical; hence, thanks to Lemma 1

$$[\![collapse(push_k(s))]\!]_e^{q_i} = [\![collapse(s)]\!]_e^{q_i} = C^{q_i}.$$

If $e > k$, $collapse(s) = collapse(push_k(s))$; hence we also have

$$[\![collapse(push_k(s))]\!]_e^{q_i} = C^{q_i}.$$

Next, for $j < k$, $pop_j(push_k(s))$ and $pop_j(s)$ are top_{j+1}-identical; hence, thanks to Lemma 1, $[\![pop_j(push_k(s))]\!]_j^{q_i} = [\![pop_j(s)]\!]_j^{q_i} = T_j^{q_i}$. Further we have $pop_k(push_k(s)) = s$; hence $[\![pop_k(push_k(s))]\!]_k^{q_i} = \mathcal{F}_{q_i}^{\gamma,e}\ C^{\overline{q}}\ T_1^{\overline{q}}\ \cdots\ T_k^{\overline{q}}$ for every $1 \leq i \leq m$. Finally, for every $j > k$, we have $pop_j(push_k(s)) = pop_j(s)$; hence $[\![pop_j(push_k(s))]\!]_j^{\overline{q}} = T_j^{\overline{q}}$. Therefore we have

$$[\![push_k(s)]\!]_n^q = \mathcal{F}_q^{\gamma,e}\ C^{\overline{q}}\ T_1^{\overline{q}}\ \cdots\ T_{k-1}^{\overline{q}}\ (\mathcal{F}_{q_1}^{\gamma,e} C^{\overline{q}}\ T_1^{\overline{q}} \cdots T_k^{\overline{q}})\ \ \cdots\ \ (\mathcal{F}_{q_m}^{\gamma,e} C^{\overline{q}}\ T_1^{\overline{q}} \cdots T_k^{\overline{q}})\ \ T_{k+1}^{\overline{q}}\ \cdots\ T_n^{\overline{q}}$$

On the other hand, it follows syntactically from the definition of $\xrightarrow{\theta}$ that the right-hand side of the preceding equation is the term t such that $[\![s]\!]_n^p \xrightarrow{\theta} t$. Hence one has $[\![s]\!]_n^p \xrightarrow{\theta} [\![push_k(s)]\!]_n^q$.

- Assume $op = pop_k$. By definition we have

$$[\![pop_k(s)]\!]_n^q \quad = \quad \mathcal{F}_q^{\gamma',e'} [\![collapse(pop_k(s))]\!]_e^{\overline{q}} [\![pop_1(pop_k(s))]\!]_1^{\overline{q}} \cdots [\![pop_n(pop_k(s))]\!]_n^{\overline{q}}$$

where the top_1 element in $pop_k(s)$ is a γ' and has an $(e'+1)$-link. It follows that

$$[\![pop_k(s)]\!]_n^q = [\![pop_k(s)]\!]_k^q [\![pop_{k+1}(pop_k(s))]\!]_{k+1}^{\overline{q}} \cdots [\![pop_n(pop_k(s))]\!]_n^{\overline{q}}$$

For every $j > k$, we have $pop_j(pop_k(s)) = pop_j(s)$, hence $[\![pop_j(pop_k(s))]\!]^{\overline{q}}_j = T^{\overline{q}}_j$. Therefore we have $[\![pop_k(s)]\!]^q_n = T^q_k\ T^{\overline{q}}_{k+1} \cdots T^{\overline{q}}_n$. On the other hand, it follows syntactically from the definition of $\xrightarrow{\theta}$ that the right-hand side of the preceding equation is the term t such that $[\![s]\!]^p_n \xrightarrow{\theta} t$. Hence one has $[\![s]\!]^p_n \xrightarrow{\theta} [\![pop_k(s)]\!]^q_n$.

- Assume $op = collapse$. By definition we have

$$[\![collapse(s)]\!]^q_n = \mathcal{F}^{\gamma',e'}_q [\![collapse(collapse(s))]\!]^{\overline{q}}_e \\ [\![pop_1(collapse(s))]\!]^{\overline{q}}_1 \cdots [\![pop_n(collapse(s))]\!]^{\overline{q}}_n$$

where the top_1 element in $collapse(s)$ is γ' and has an $(e'+1)$-link. Equivalently, one has

$$[\![collapse(s)]\!]^q_n \ = \ [\![collapse(s)]\!]^q_e [\![pop_{e+1}(collapse(s))]\!]^{\overline{q}}_{e+1} \ \cdots \ [\![pop_n(collapse(s))]\!]^{\overline{q}}_n$$

For every $j > e$ we have $pop_j(collapse(s)) = pop_j(s)$, hence $[\![pop_j(collapse(s))]\!]^{\overline{q}}_j = T^{\overline{q}}_j$. Therefore we have $[\![collapse(s)]\!]^q_n = C^q\ T^{\overline{q}}_{e+1} \cdots T^{\overline{q}}_n$. On the other hand, it follows syntactically from the definition of $\xrightarrow{\theta}$ that the right-hand side of the preceding equation is the term t such that $[\![s]\!]^p_n \xrightarrow{\theta} t$. Hence one has $[\![s]\!]^p_n \xrightarrow{\theta} [\![collapse(s)]\!]^q_n$.

□

We define a relation $\sim$ between configurations of $\mathcal{A}$ and ground-type terms generated from symbols from the set

$$\mathcal{N} = \{\mathcal{F}^{\gamma,e}_p \mid \gamma \in \Gamma,\ 1 \le e \le n,\ p \in Q\} \cup \{\mathrm{Void}_i \mid 1 \le i \le n\},$$

defined by $(p, s) \sim [\![s]\!]^p_n$. Then $\sim$ is a bisimulation.

5.4 The recursion scheme $G_{\mathcal{A}}$ determined by a CPDA $\mathcal{A}$

Fix some integer $d \ge 1$ and let $[d]$ denote $\{1, \ldots, d\}$. Fix an n-CPDA $\mathcal{A} = \langle\, [d] \cup \{\varepsilon\}, \Gamma, Q, \delta, q_1 \,\rangle$ and a function $\rho : Q \times \Gamma \to \Sigma$, and let $Q = \{q_1, \cdots q_m\}$. Let t be the tree generated by $\mathcal{A}$ and ρ as defined in Section 3.3.

We define from $\mathcal{A}$ and ρ an order-n recursion scheme whose value tree is t. The main idea here is to rely on the previous term representation of configurations of $\mathcal{A}$. Indeed, what we did so far was to define an $([d] \cup \{\varepsilon\})$-edge-labelled transition system whose elements are finite terms of ground type and to prove that it is bisimilar (in the usual sense) with Graph($\mathcal{A}$). Hence, it suffices to design a recursion scheme that mimics the dynamics of the previous term-rewrite system, *i.e.* such that its value tree is the tree obtained from the previous transition system by unfolding, contracting the ε-transitions, and labelling (according to the head non terminal).

Definition 3. *The order-n* ***recursion scheme determined by $\mathcal{A}$ and ρ*** *is defined to be $G_{\mathcal{A},\rho} = \langle\, \Sigma, \mathcal{N}, \mathcal{R}, S \,\rangle$ (written $G_{\mathcal{A}}$ if ρ is clear) where*

$$\mathcal{N} = \{\mathcal{F}^{\gamma,e}_p \mid \gamma \in \Gamma,\ 1 \le e \le n,\ p \in Q\} \cup \{\mathrm{Void}_i \mid 1 \le i \le n\}$$

consists of those non-terminals as introduced in Section 5.1, and the rules in $\mathcal{R}$ are as follows

$$S \longrightarrow \rho(q_1, \bot) \qquad \text{if } ar(\rho(q_1, \bot)) = 0$$

$$S \longrightarrow \mathcal{F}_{q_1}^{\bot,1}\ \overline{\mathrm{Void}_1}\ \cdots \overline{\mathrm{Void}_n} \qquad \text{otherwise}$$

and

$$\mathcal{F}_p^{\gamma,e}\ \overline{\Phi}\ \overline{\Psi_1} \cdots \overline{\Psi_n} \longrightarrow \Xi_\theta \qquad \text{if } \delta(p,\gamma,\varepsilon) = \theta \text{ is defined}$$

$$\mathcal{F}_p^{\gamma,e}\ \overline{\Phi}\ \overline{\Psi_1} \cdots \overline{\Psi_n} \longrightarrow \rho(p,\gamma)\ \Xi_{\theta_1} \cdots \Xi_{\theta_r} \qquad \text{otherwise, where } r = ar(\rho(p,\gamma)) \text{ where}$$

$$\theta_i = \delta(p,\gamma,i) \text{ for } i = 1, \ldots, r.$$

Note that in the definition, we need to distinguish those states p and stack symbols γ where $\{a \in [d] \cup \{\varepsilon\} \mid (p,\gamma,a) \in \mathsf{Dom}(\delta)\} = \varnothing$. Indeed, one still needs to produce a terminal for them as they correspond to productive leaves in the tree obtained from Graph($\mathcal{A}$) by unfolding and contracting the ε-transitions.

We are now in a position to state the major result of the section.

Theorem 3 (Equi-Expressivity 1)**.** *Let $\mathcal{A}$ be a tree-generating CPDA, and $G_{\mathcal{A}}$ be the recursion scheme determined by $\mathcal{A}$. Then the CPDA and the recursion scheme generate the same Σ-labelled tree.*

Proof. The proof follows from Proposition 1, the definition of $G_{\mathcal{A}}$ and the way one generates a tree from a CPDA.

The key idea here is to give a precise description of the terms t such that $S \to^*_{G_{\mathcal{A}}} t$ where $\to^*_{G_{\mathcal{A}}}$ denotes the transitive closure of $\to_{G_{\mathcal{A}}}$.

Let s and t be two finite terms of ground type. We say that s is a ***subterm*** of t if either $s = t$, or there exist $f \in \Sigma$ and $i \in \{1, \cdots, \ell\}$ such that $t = f t_1 \cdots t_\ell$ and s is a subterm of t_i. Note that, in the sequel, we implicitly distinguish two copies of a term that appears as subterm in different parts of a given term. More formally, every subterm s of a term t has a ***location***, denoted $location_t(s)$ (or simply $location(s)$ if t is clear), which is a sequence, where $\cdot$ denotes concatenation of sequences. It is defined by

$$location_t(s) := \begin{cases} \varepsilon & \text{if } s = t \\ f \cdot i \cdot location_{t_i}(s) & \text{otherwise, where } t = f\ t_1 \cdots t_\ell \text{ and} \\ & s \text{ is a subterm of } t_i \end{cases}$$

One can easily characterise those terms that can be derived from S in $\to_{G_{\mathcal{A}}}$. Indeed we have $S \to^*_{G_{\mathcal{A}}} t$ if and only if either $t = S$ or for every subterm t' of t such that $location_t(t') = f_1 \cdot a_1 \cdots f_\ell \cdot a_\ell \in (\Sigma \cdot \{1, \cdots, d\})^*$ with $\ell \geq 0$, we have $t' = [\![s]\!]^p_n$ for some configuration (p, s) in Graph($\mathcal{A}$) such that there exist a sequence $(p_0, s_0), \ldots, (p_{\ell+1}, s_{\ell+1})$ of configurations of Graph($\mathcal{A}$) and numbers $k_1, \ldots, k_{\ell+1} \geq 0$ such that

- $(p_0, s_0) = (q_1, \bot_n)$ is the initial configuration;
- $(p_0, s_0) \xrightarrow{\varepsilon^{k_1}} (p_1, s_1)$;
- $(p_i, s_i) \xrightarrow{a_i \varepsilon^{k_{i+1}}} (p_{i+1}, s_{i+1})$ for all $1 \leq i \leq \ell - 1$;

- $(p_\ell, s_\ell) \xrightarrow{a_\ell \varepsilon^{k_{\ell+1}}} (p_{\ell+1}, s_{\ell+1})$;
- $(p_{\ell+1}, s_{\ell+1}) = (p, s)$;
- $\rho(p_i, top_1(s_i)) = f_i$ for all $1 \leq i \leq \ell$.

The previous characterisation is proved directly by an induction on the number of rewrite rules applied to derive t from S: the base case is immediate, and the inductive step follows from Proposition 1.

It follows from the previous lemma and the definition of a tree generated by a CPDA that the value tree of $G_{\mathcal{A}}$ is the tree generated by $\mathcal{A}$ and ρ. □

6 From recursion schemes to CPDA

The previous section demonstrates that higher-order recursion schemes are at least as expressive as CPDAs. In this section we prove the converse. Hence, CPDAs and recursion schemes are equi-expressive. A number of related results can be found in the literature, but an exact correspondence with general recursion schemes has never been proved before. Notably, in order to establish a correspondence between recursion schemes and higher-order PDAs, Damm and Goerdt (for word languages [25, 26]) as well as Knapik, Niwiński and Urzyczyn (for labelled trees [40]), have had to impose constraints on the shape of the former (called *derived types* and *safety* respectively) and their translation techniques relied on the restrictions in a crucial way.

Our translation from recursion schemes to CPDA is novel: we transform an arbitrary order-n recursion scheme G to an order-n *collapsible pushdown automaton* $\mathcal{A}_G$ that computes the *traversals* over the computation tree $\lambda(G)$ (in the sense of [53, 54]). The game-semantic interpretation of G is an *innocent strategy* (in the sense of [35]), which coincides with the *value tree* $[\![G]\!]$ of G, so that paths in the value tree are plays of the strategy. Traversals over the computation tree are just (appropriate representations of) *uncoverings* [35] of paths in the value tree.

6.1 Long transform, graph representing a recursion scheme, traversals

We first introduce several concepts we need for the rest of the section.

We write $[n]$ as a shorthand for $\{1, \cdots, n\}$ and $[n]_0$ for $\{0, \cdots, n\}$. Fix a ranked alphabet Σ. Typically[4] $\mathsf{Dir}(f) = [ar(f)]$ and we always have $|\mathsf{Dir}(f)| = ar(f)$ for each Σ-symbol f.

We recall the long transform of a recursion scheme as introduced in [54]. Fix a recursion scheme G. Rules of the new recursion scheme $\overline{G}$ (which, we shall see, can be regarded as order 0) are obtained from those of G by applying the following four operations in turn, which is called ***long transform***. For each G-rule:

1. *Expand the right-hand side to its η-long form. I.e.* we hereditarily η-expand every subterm – even if it is of ground type – provided it occurs in an *operand position*. Note that each term $s \in \mathcal{T}(\Sigma \cup \mathcal{N} \cup \{\xi_1, \cdots, \xi_l\})$ can be written uniquely as $\dagger\, s_1 \cdots s_m$ where $\dagger$ is either a variable (*i.e.* some ξ_j) or a non-terminal or a terminal. Suppose $\dagger\, s_1 \cdots s_m : (A_1, \cdots, A_n, o)$. We define

$$\ulcorner \dagger\, s_1 \cdots s_m \urcorner \quad = \quad \boldsymbol{\lambda}\overline{\varphi}.\dagger \ulcorner s_1 \urcorner \cdots \ulcorner s_m \urcorner \ulcorner \varphi_1 \urcorner \cdots \ulcorner \varphi_n \urcorner$$

[4] The only exception is the symbol $@_A$ of the auxiliary alphabet Λ_G, where we have $\mathsf{Dir}(@_A) = [ar(@_A) - 1]_0$.

where $\overline{\varphi}$ is a list $\varphi_1 \cdots \varphi_n$ of (fresh) pairwise-distinct variables (which is a null list iff $n = 0$) of types $A_1, \cdots, A_n$ respectively, none of which occurs free in $\dagger \ulcorner s_1 \urcorner \cdots \ulcorner s_m \urcorner$.

For example the η-long form of $g\, a : o$ is $\boldsymbol{\lambda}.g\,(\boldsymbol{\lambda}.a)$; we shall see that the "dummy lambda-abstraction"[5] $\boldsymbol{\lambda}.a$ (that binds a *null* list of variable) plays a useful rôle in the syntactic representation of the game semantics of a recursion scheme.

2. *Insert long-apply symbols* $@_A$: Replace each ground-type subterm of the shape $D\, e_1 \cdots e_n$, where $D : (A_1, \ldots, A_n, o)$ is a non-terminal and $n \geq 1$ (*i.e.* D has order at least 1), by $@_A\, D\, e_1 \cdots e_n$ where $A = ((A_1, \ldots, A_n, o), A_1, \ldots, A_n, o)$ and $@_A : A$. In the following, we shall often omit the type tag A from $@_A$, as it is uniquely determined by the respective types of $D, e_1, \ldots, e_n$.

3. *Curry the rewrite rule. I.e.* we transform the rule $F\, \varphi_1 \cdots \varphi_n \;\rightarrow\; \boldsymbol{\lambda}.e'$ to

$$F \;\rightarrow\; \boldsymbol{\lambda}\varphi_1 \cdots \varphi_n.e'.$$

In case $n = 0$, note that the curried rule has the form $F \;\rightarrow\; \boldsymbol{\lambda}.e'$.

4. *Rename bound variables afresh*, so that any two variables that are bound by different lambdas have different names.

Exampl 8. *We revisit the recursion scheme of Example 1 and illustrate the long transform:*

$$G : \begin{cases} S & \rightarrow & H\, a \\ H\, z & \rightarrow & F\,(g\, z) \\ F\, \varphi & \rightarrow & \varphi\,(\varphi\,(F\, h)) \end{cases} \quad \mapsto \quad \overline{G} : \begin{cases} S & \rightarrow & \boldsymbol{\lambda}.@\, H\,(\boldsymbol{\lambda}.a) \\ H & \rightarrow & \boldsymbol{\lambda}z.@\, F\,(\boldsymbol{\lambda}y.g\,(\boldsymbol{\lambda}.z)\,(\boldsymbol{\lambda}.y)) \\ F & \rightarrow & \boldsymbol{\lambda}\varphi.\varphi\,(\boldsymbol{\lambda}.\varphi\,(\boldsymbol{\lambda}.@\, F\,(\boldsymbol{\lambda}x.h\,(\boldsymbol{\lambda}.x)))) \end{cases}$$

For instance, the right hand side of the third rule is $\boldsymbol{\lambda}.\varphi\,(\boldsymbol{\lambda}.\varphi\,(\boldsymbol{\lambda}.F\,(\boldsymbol{\lambda}x.h\,(\boldsymbol{\lambda}.x))))$ *after the first step, and* $\boldsymbol{\lambda}.\varphi\,(\boldsymbol{\lambda}.\varphi\,(\boldsymbol{\lambda}.@\, F\,(\boldsymbol{\lambda}x.h\,(\boldsymbol{\lambda}.x))))$ *after the second step.*

For every recursion scheme G, the system of transformed rules in $\overline{G}$ defines an order-0 recursion scheme – called the ***long transform*** of G – with respect to an enlarged ranked alphabet Λ_G, which is Σ augmented by certain variables and lambdas (of the form $\boldsymbol{\lambda}\overline{\xi}$ which is a short hand for $\boldsymbol{\lambda}\xi_1 \cdots \xi_n$ where $n \geq 0$) *but regarded as terminals.* The alphabet Λ_G is a finite subset of the set

$$\underbrace{\Sigma \;\cup\; \mathit{Var} \;\cup\; \{\, @_A \mid A \in \mathit{ATypes} \,\}}_{\textbf{\textit{Non-lambdas}}} \quad \cup \quad \underbrace{\{\, \boldsymbol{\lambda}\overline{\xi} \mid \overline{\xi} \subseteq \mathit{Var} \,\}}_{\textbf{\textit{Lambdas}}}$$

where *ATypes* is the set of types of the shape $((A_1, \cdots, A_n, o), A_1, \cdots, A_n, o)$ with $n \geq 1$. We rank the symbols in Λ_G as follows:

- variable symbol $\varphi : (A_1, \cdots, A_n, o)$ in *Var* has arity n
- long-apply symbol $@_A$ where $A = ((A_1, \cdots, A_n, o), A_1, \cdots, A_n, o)$ has arity $n + 1$

[5] To our knowledge, Colin Stirling was the first to use a tree representation of lambda terms in which "dummy lambdas" are employed; see his paper [68]. Motivated by property-checking games in Verification, he has introduced a game that is played over such trees as a characterisation of higher-order matching [69].

- lambda symbol $\boldsymbol{\lambda}\overline{\xi}$ has arity 1, for every list of variables $\overline{\xi} \subseteq \mathit{Var}$.

Further, for $f \in \Lambda_G$, we define

$$\mathsf{Dir}(f) \;=\; \begin{cases} [ar(@_A) - 1]_0 & \text{if } f = @_A \\ [ar(f)] & \text{otherwise} \end{cases}$$

For technical reasons (to be clarified shortly), the leftmost child of an @-labelled node α is in direction 0 (*i.e.* it is α's 0-child); for all other nodes, the leftmost child is in direction 1. The *non-terminals* of $\overline{G}$ are exactly those of G, except that each is assigned a new type, namely, o. We can now define the ***computation tree*** $\lambda(G)$ to be the value tree $[\![\,\overline{G}\,]\!]$ of the order-0 recursion scheme $\overline{G}$. It follows that $\lambda(G)$ is a regular tree[6].

A ***Λ-labelled rooted deterministic digraph*** (or DDG, for short) is a quadruple

$$\mathcal{K} = \langle\, V, E, l, v_0 \,\rangle$$

where $\langle\, V, E \,\rangle$ is a finite digraph vertex-labelled by the function $l : V \longrightarrow \Lambda$ with Λ a ranked alphabet, such that each vertex $v \in V$ has as many successors as the arity of $l(v)$, and each of these successors are ordered; and $v_0 \in V$ is a distinguished vertex called the *root*. We denote by $E_i(v)$ the unique i-th successor of v for $i = 1, \ldots, ar(l(v))$.

It is easy to see that every finite Λ-labelled tree can be presented as a DDG.

The ***unfolding*** of $\mathcal{K}$ is the Λ-labelled ranked tree $t : \mathsf{Dom}(t) \longrightarrow \Lambda$ such that $\mathsf{Dom}(t)$ is the set of finite paths in $\mathcal{K}$ starting from the root v_0, and $t(v_0 \cdots v_k) = l(v_k)$ where the i-th child (when defined) of node $v_1 \cdots v_k$ is $v_1 \cdots v_k E_i(v_k)$. This definition (canonically) associates vertices in $\mathcal{K}$ with nodes in t: the node ε is mapped to v_0, and the node $v_1 \cdots v_k$ ($k \geq 1$) is mapped to v_k. This map extends to an association of node sequences in t with vertex sequences in $\mathcal{K}$. When restricted to paths in t and paths in $\mathcal{K}$ starting from the root, we obtain a bijection.

Fix a higher-order recursion scheme G and an associated long transform $\overline{G}$. We define the ***HORS graph*** $\mathsf{Gr}(\boldsymbol{G})$ to be the Λ_G*-labelled DDG determined by* G

$$\mathsf{Gr}(G) = \quad \langle\, V, \quad E \subseteq V \times V, \quad \lambda_G : V \longrightarrow \Lambda_G, \quad v_0 \in V \,\rangle$$

that is obtained by the following procedure:

1. First we define the ranked alphabet $\Lambda_{G,\overline{G}} = \Lambda_G \cup \mathcal{N}_{\overline{G}}$ where each symbol in $\mathcal{N}_{\overline{G}}$ (*i.e.* a non-terminal of $\overline{G}$) is given arity 0.

2. For each $\overline{G}$-rule (say) $F \;\to\; \boldsymbol{\lambda}\varphi_1 \cdots \varphi_n.e$, the corresponding $\Lambda_{G,\overline{G}}$-labelled DDG

$$\mathcal{D}_F \quad = \quad \langle\, V_F, \quad E^F \subseteq V_F \times V_F, \quad l_F : V_F \longrightarrow \Lambda_{G,\overline{G}}, \quad rt_F \,\rangle$$

given by the $\Lambda_{G,\overline{G}}$-labelled tree that is determined by the right-hand side of the rule, namely, $\boldsymbol{\lambda}\varphi_1 \cdots \varphi_n.e$. In particular, rt_F is the root of that tree and we have $l_F(rt_F) = \boldsymbol{\lambda}\varphi_1 \cdots \varphi_n$ with reference to the rule F given above.

[6] An infinite tree is *regular* if and only if it contains finitely many different infinite subtrees. Equivalently, an infinite tree is regular if it can be obtained by unfolding a finite directed graph.

3. First for each F in $\mathcal{N}_{\overline{G}}$ we define

$$\mathcal{E}_F = \bigcup_{H \in \mathcal{N}_{\overline{G}}} l_H^{-1}(\{ F \}) \qquad \text{and} \qquad \mathcal{E} = \bigcup_{\mathcal{F} \in \mathcal{N}_{\overline{G}}} \mathcal{E}_{\mathcal{F}}$$

Then $\mathsf{Gr}(G)$ is intuitively obtained by taking the disjoint union of the underlying digraphs of $\mathcal{D}_F$ and then merging with rt_F all those vertices that belong to same $\mathcal{E}_F$, as F ranges over $\mathcal{N}_{\overline{G}}$.

Formally we have the following.

- The vertices V of $\mathsf{Gr}(G)$ are defined by

$$V = \bigcup_{F \in \mathcal{N}_{\overline{G}}} V_F \setminus \bigcup_{F \in \mathcal{N}_{\overline{G}}} \mathcal{E}_F$$

- The root v_0 of $\mathsf{Gr}(G)$ is rt_S, where S is the start symbol of $\overline{G}$.
- The vertex-labels are defined by

$$\lambda_G(v) \quad = \quad \begin{cases} l_F(rt_F) & \text{if } v = \mathcal{E}_F \text{ for some } F \in \mathcal{N}_{\overline{G}} \\ l_H(v) & \text{otherwise, where } v \text{ is a vertex in } V_H \end{cases}$$

- The edges E of $\mathsf{Gr}(G)$ are defined by

$$E = (\bigcup_{F \in \mathcal{N}_{\overline{G}}} E_F) \quad \setminus \quad \{(v, v') \mid v \in \mathcal{E} \text{ or } v' \in \mathcal{E}\} \quad \cup$$
$$\bigcup_{F' \in \mathcal{N}_{\overline{G}}} \{(rt_F, v') \mid (v, v') \in E_{F'} \text{ and } v \in \mathcal{E}_F\} \cup \{(v', rt_F) \mid (v', v) \in E_{F'} \text{ and } v \in \mathcal{E}_F\}$$

and the edge-labels of $\mathsf{Gr}(G)$ are inherited from the edge-labels of the component DDGs $\mathcal{D}_F$ according to how vertices and edges where merged.

In the following, we shall only concern ourselves with the connected component of $\mathsf{Gr}(G)$ that contains the root node (and assume that $\mathsf{Gr}(G)$ *is* that connected component)[7]. It follows from the definitions that unfolding $\mathsf{Gr}(G)$ gives the computation tree $\lambda(G)$.

Exampl 9. *We revisit the recursion scheme of examples 1 and 8. The graph* $\mathsf{Gr}(G)$ *is given in Figure 2.*

Fix a HORS graph $\mathsf{Gr}(G) = \langle V, E, \lambda_G, v_0 \rangle$. We shall call a vertex of $\mathsf{Gr}(G)$ ***prime*** just if it is the 0-child[8] of a @-labelled vertex. By construction, a prime vertex is labelled by a lambda. We define the ***depth*** of a vertex to be the length of the shortest path from the root to the vertex (so that the root has depth 0). Let u be a vertex. We define $\mathsf{pred}(u) = \{ u' \in V : (u', u) \in E \}$ *i.e.* the set of *predecessors* of u. For every vertex u labelled by a variable φ_i (say), its ***binder***, written $binder(u)$, is the vertex that is labelled $\boldsymbol{\lambda}\overline{\varphi}$, where $\overline{\varphi}$ is a list of variables that contains

[7]Note that if $\mathsf{Gr}(G)$ contains several connected components, some rules in G are never used to produce $[\![G]\!]$.

[8]The leftmost child of a @-labelled vertex is the latter's 0-child (*i.e.* the child is at the end of a 0-labelled edge); the leftmost child of any other vertex is a 1-child.

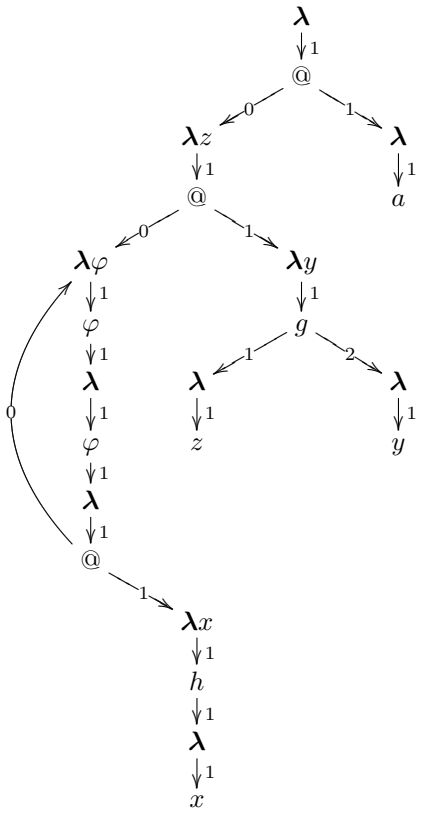


Figure 2: The graph determined by the order-2 recursion scheme from examples 1 and 8.

φ_i. (Since bound variables are renamed to prevent any clash in the construction of $\overline{G}$, every variable vertex in $\mathsf{Gr}(G)$ has a unique binder.) We say that u is the *i-parameter* of $binder(u)$ just if φ_i is the ith-item of the list $\overline{\varphi}$. The ***span*** of the variable vertex u is defined to be the depth of u minus the depth of $binder(u)$.

We note the following features of HORS graphs:

(i) Except the root and possibly some prime vertices, every vertex u has a unique predecessor v. For $j > 0$, if u is the j-child of v, we say that u is a j-child. If u is prime then it is the 0-child of all its predecessors and we then say that u is a 0-child. Indeed, a vertex is a 0-child if and only if it is prime.

(ii) For every vertex u, there is a unique shortest path from $binder(u)$ to u, and this path does not contain any prime vertex.

For convenience, and whenever it is safe to do so, we shall confuse a vertex u with its Λ_G-label $\lambda_G(u)$.

We now define several notions regarding the computation tree. For simplicity, we shall refer to a node labelled by some lambda (*resp.* variable) as a *lambda node* (*resp.* a *variable node*).

The notion of binder can also be defined for the computation tree. Indeed, let n be some node in $\lambda(G)$ labelled by a variable ξ. We say that n is ***bound*** by the node n' (equivalently

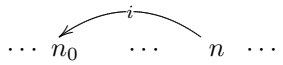
that n' is the ***binder*** of n) just in case n' is the largest prefix of n that is labelled by a lambda symbol $\lambda\overline{\xi}$ for some list $\overline{\xi}$ that contains ξ.

Binders allow us to define a binary relation $\vdash_i$ over the set of nodes of $\lambda(G)$, called ***enabling*** (we read $n \vdash_i n'$ as "n i-enables n'", or "n' is i-enabled by n"), as follows.

- Every lambda node, except the root, is *i-enabled* by its parent node in $\lambda(G)$, where the former is the i-child of the latter.
- Every variable node (labelled by some ξ_i, say) is *i-enabled* by its binder (labelled by some $\overline{\xi}$, say) where ξ_i is the i-th element of the list $\overline{\xi}$.

We say that a node of $\lambda(G)$ is *initial* if it is not enabled by any node. It follows from the definition that the initial nodes are the root-node (necessarily labelled by the lambda symbol λ), and all nodes labelled by a long-apply or a Σ-symbol.

Enabling permits us to define the notion of *justified sequence* over the computation tree. A ***justified sequence*** over $\lambda(G)$ is a possibly infinite, lambda / non-lambda alternating sequence of nodes that satisfies the *pointer condition*: Each non-initial node n that occurs in it has a pointer to some earlier node-occurence n_0 in the sequence such that $n_0 \vdash_i n$ for some i. We say that the node-occurence n is ***justified*** by the node-occurence n_0 in the sequence. We use the notation



to mean that n points to n_0 and that $n_0 \vdash_i n$ holds. We say that n is *i-justified* by n_0, or n has a *i-pointer* to n_0 in the justified sequence. Let us stress that a justified sequence need not be a path. [9]

Let $\mathbf{t}$ be a justified sequence and let n be some occurrence of a node in $\mathbf{t}$. Then we write $\mathbf{t}_{\leq n}$ (*resp.* $\mathbf{t}_{<n}$) to mean the prefix of $\mathbf{t}$ truncated at and including (*resp.* excluding) the node n.

We are now ready to introduce *traversals*. ***Traversals*** over the computation tree $\lambda(G)$ are justified sequences of nodes defined by induction over the rules given in Table 2.

A remark on rule (Lam). If n' is a variable then it should point to the binder. But there may be several occurrences of the binder, and this is what the P-view is for: it selects an occurrence via the pointer from the variable in question.

Remark 5. *The pointers in a traversal over any computation tree $\lambda(G)$ are uniquely reconstructible from the underlying sequence of nodes and their respective labels; thus pointers are not an additional structure imposed on the underlying sequence. However it is convenient (e.g. in the definition of P-view below) to define traversals as sequences equipped with pointers. Another advantage of pointers is that they help to clarify the correspondence between traversals and* interaction sequences *(that arise in the construction of the game semantics of the recursion scheme in question).*

Note that the only rule in Table 2 that can lead to extending a traversal in a non unique way is the rule *(Sig)* that allows $ar(f)$ possible extensions. Traversals define an infinite rooted

[9] Justified sequences were first introduced to represent plays in dialogue games between two players, O and P [35]. A play is a certain sequence of alternating O-moves and P-moves. A player may only make a given move m provided the move that enables it, m' (say), has already been played; and if so, this situation is represented by a pointer from m to m'.

Table 2: Rules for defining traversals.

(Root) The singleton sequence, comprising the root node ε, is a traversal.

(App) If $t\ n$ is a traversal for some sequence t and some node n labelled by @, so is $t\ n \xleftarrow{0} n'$ for some node n'. Note that n' is always a lambda node.

(Sig) If $t\ n$ is a traversal for some sequence t and some node n labelled by a Σ-symbol f, so is $t\ n \xleftarrow{i} n'$ for each $1 \leq i \leq ar(f)$ and some node n'. Note that n' is always labelled by a dummy $\boldsymbol{\lambda}$.

(Var) If $t\ n\ n' \xleftarrow{i} \cdots\ n''$ is a traversal for some sequence t and some lambda node n' (labelled by some $\boldsymbol{\lambda}\overline{\xi}$) and some variable node n'' (labelled by ξ_i, the i-th variable in $\overline{\xi}$), so is $t\ n \xleftarrow{i} n' \xleftarrow{i} \cdots\ n'' \xleftarrow{i} n'''$ for each node n'''. Note that n''' is always a lambda node.

(Lam) If $t\ n$ is a traversal for some sequence t and some lambda node n, so is $t\ n\ n'$ where n' is the 1-child of n. By a straightforward induction, $\ulcorner t\ n \urcorner$ is a path in the tree $\lambda(G)$; if n' is labelled by a variable (as opposed to @) then its pointer in $t\ n\ n'$ is determined by the condition that $\ulcorner t\ n\ n' \urcorner$ is a path in $\lambda(G)$.

deterministic digraph[10]

$$Tr(G) = \langle\, V, E \subseteq V \times (\mathsf{Dir}(\Sigma) \cup \{\varepsilon\}) \times V, l : V \to (\Sigma \cup \{\sharp\}), v_0 \,\rangle$$

where

- V is the set of all traversals over $\lambda(G)$.
- $(v, \varepsilon, v') \in E$ iff v' is obtained from v by applying one of the following rules: *(App)*, *(Var)* or *(Lam)*.
- $(v, i, v') \in E$ iff v' is obtained from v by applying rule *(Sig)* with parameter i.
- $l(v) = \lambda(G)(n)$ if v ends by a node n labelled by a terminal, and $l(v) = \sharp$ otherwise.
- The root v_0 is the traversal ε.

Note that $Tr(G)$ is a deterministic tree. By taking its ε-closure as explained below, we obtain a Σ-labelled ranked tree. More precisely, the ***traversal tree*** of G, denoted $\boldsymbol{TrTree(G)}$, is obtained as follows from $Tr(G)$. For every $i \in \mathsf{Dir}(\Sigma)$, add an i-labelled edge from v_1 to v_2 whenever there is a path from v_1 to v_2 labelled by a word that matches $i\varepsilon^*$ (note that we treat ε as a standard letter), and there is no outgoing ε-labelled edge from v_2; for every $i \in \mathsf{Dir}(\Sigma)$,

[10] In fact we have a small abuse of terminology and notation here as in our original definition of a DDG we do not allow ε-labelled edges which we do here only for those edges outgoing an $\sharp$-labelled vertex. For ease of writing we also make the edge label explicit by describing edges as elements of $V \times \{\mathsf{Dir}(\Sigma) \cup \{\varepsilon\}\} \times V$.

whenever there is an edge from v_1 to a node v_2 from which there is an infinite path made only of ε-labelled edges, create a new vertex $v_1^{i,\perp}$ labelled by $\perp$ and add an i-labelled edge from v_1 to $v_1^{i,\perp}$; then remove any vertex that is the source of an ε-labelled edge and remove any ε-labelled edge. In case the resulting object is empty, simply replace it by a tree made only of a root labelled by $\perp$. Note that the resulting object is always a deterministic, ε-free, Σ-labelled tree (indeed, $\sharp$-labelled nodes, *i.e.* those that are not labelled by terminal, as well as ε-labelled edges have all been removed). Define the root as the (unique) node that is reachable in $Tr(G)$ by a (possibly empty) sequence of ε-labelled edges, and not the source of an ε-labelled edge. Call $TrTree(G)$ the resulting tree.

In the sequel, to stick to our original definition of ranked-trees, we regard $TrTree(G)$ as the Σ-labelled ranked tree whose domain is the set of words in $\mathsf{Dir}(\Sigma)$ that labels finite path from the root, and where a node n is labelled by $l(v)$ where v is the (unique) node reached by following from the root the path labelled by n.

It turns out that the value tree and the traversal tree are equal [56]:

Theorem 4 (Correspondence Theorem)**.** *For every recursion scheme G, we have $[\![G]\!] = TrTree(G)$.*

We finally define the *P-view* of a justified sequence. The ***P-view*** $\ulcorner t \urcorner$ of a justified sequence t is a subsequence defined by recursion as follows:

- $\ulcorner \boldsymbol{\lambda} \urcorner = \boldsymbol{\lambda}$ for a dummy lambda $\boldsymbol{\lambda}$.
- $\ulcorner\ t\ n' \overset{i}{\leftarrow} \cdots\ n \urcorner = \ulcorner t \urcorner\ n' \overset{i}{\leftarrow} n$ whenever n is a lambda node (hence n' is a non-lambda node). Node n' is either @ or a signature symbol, and in the latter case the lambda is dummy.
- $\ulcorner t\ n \urcorner = \ulcorner t \urcorner\ n$ whenever n is a non-lambda node.

In the second clause above, if in $t\ n' \overset{i}{\leftarrow} \cdots\ n$ the non-lambda node n' has a pointer to some node-occurence l (say) in t, and if l appears in $\ulcorner t \urcorner$, then in $\ulcorner t \urcorner\ n' \overset{i}{\leftarrow} n$ the node n' is defined to point to l; otherwise n' has no pointer. Similarly, in the third clause above, if in $t\ n$ the non-lambda node n has a pointer to some node-occurence l (say) in t and if l appears in $\ulcorner t \urcorner$, then in $\ulcorner t \urcorner\ n$ the node n is defined to point to l; otherwise n has no pointer.

In is easy to see that the P-view of a justified sequence is always lambda/non-lambda alternating, that may not necessarily satisfy the pointer condition. When applied to traversals, the P-view has some nice properties.

Proposition 2. *[53, Lemma 2][54, Proposition 6] Let t be a finite traversal over a computation tree $\lambda(G)$. Then the following hold.*

- *$\ulcorner t \urcorner$ is a well-defined justified sequence.*
- *$\ulcorner t \urcorner$ is a path in the computation tree $\lambda(G)$ from the root to the last node in t.*

Traversals (and related concepts) were defined with respect to the computation tree $\lambda(G)$. As $\lambda(G)$ is obtained by unfolding the HORS graph $\mathsf{Gr}(G)$ we can associate with every sequence of nodes in $\lambda(G)$ a unique sequence of vertices in $\mathsf{Gr}(G)$. This mapping, when restricted to (the sequences of nodes underlying) traversals over the computation tree $\lambda(G)$, is injective. Hence, depending on the context, we may see traversals either as sequences of nodes in $\lambda(G)$ or as sequences of vertices in $\mathsf{Gr}(G)$ (in this case, it is easy to reconstruct the corresponding traversal over $\lambda(G)$). Note that traversals over $\mathsf{Gr}(G)$ could equivalently be defined by stating the rules in Table 2 in the framework of $\mathsf{Gr}(G)$.

6.2 $\mathsf{CPDA}(G)$ — the CPDA determined by a recursion scheme G

As stated in the previous section (Correspondence Theorem), traversals provide an alternative way to describe the value tree of a given scheme G. We will now define a CPDA, $\mathsf{CPDA}(G)$, and will show that its transition graph $\mathrm{Graph}(\mathsf{CPDA}(G))$ is trace-equivalent with $Tr(G)$. As a byproduct, unfolding the ε-closure of $\mathrm{Graph}(\mathsf{CPDA}(G))$ leads to the same tree as $TrTree(G) = \lambda(G)$, equivalently $\mathsf{CPDA}(G)$ generates the same tree as G.

Fix an order-n recursion scheme G and the HORS graph

$$\mathsf{Gr}(G) \quad = \quad \langle\, V,\ E \subseteq V \times V,\ \lambda_G : V \longrightarrow \Lambda_G,\ v_0 \in V \,\rangle$$

determined by it. Note that G is not assumed to be homogeneously typed, and hence, not necessarily safe

Remark 6. *For convenience, in the definition of the transform $\mathsf{CPDA}(G)$, we shall write $push_1^{a,1}$ as $push_1^a$, effectively ignoring the 1-link (to the preceding stack symbol). This is harmless since 1-links are guaranteed not to feature in any of collapse operations of the transform $\mathsf{CPDA}(G)$.*

Definition 4. *The transform $\mathsf{CPDA}(G)$ is an n-CPDA with a single dummy control state (that we omit from now for simplicity) that has the set V of nodes as the stack alphabet. The initial configuration is the n-stack $[\cdots[\perp v_0]\cdots]$ i.e. $push_1^{v_0} \perp_n$, where v_0 is the root of $\mathsf{Gr}(G)$. Let u range over the stack symbols of $\mathsf{CPDA}(G)$. For ease of explanation, we define the transition map δ as a function that takes a node $u \in V$ to a sequence of stack operations (in particular, this allows us to have a single control state), by a case analysis of the label (from Λ_G) of u. The definition is presented in Table 3.*

Remark 7. *The transformation is radically different from the compilation method of Knapik* et al. *[40, 41]. To date, it is not known whether the approach in [41] is extendable to non-homogeneously typed recursion schemes of order 2. More generally, it is not known whether the method is extendable to arbitrary recursion schemes of all finite orders.*

6.3 $\mathrm{Graph}(\mathsf{CPDA}(G))$ and $Tr(G)$ are trace equivalent

We first recall the standard notion of trace equivalence, that we state here in the specific case of labelled rooted deterministic digraphs. Let $\mathcal{K}_1 = \langle\, V_1, E_1 \subseteq V_1 \times \Pi \times V_1, l : V_1 \to \Lambda, r_1 \,\rangle$ and $\mathcal{K}_2 = \langle\, V_2, E_2 \subseteq V_2 \times \Pi \times V_2, l : V_2 \to \Lambda, r_1 \,\rangle$ be two graphs with the same alphabet for labelling vertices (*resp.* edges). We say that $\mathcal{K}_1$ and $\mathcal{K}_2$ are ***trace-equivalent*** just if for every $x \in \{1, 2\}$, for every path in $\mathcal{K}_x$ that starts from the root r_x, there is a (unique) corresponding path in $\mathcal{K}_{\overline{x}}$ (here $\overline{x} = 2$ if $x = 1$ and $\overline{x} = 1$ otherwise) with the same label in Π^*. Moreover the terminal vertices of both paths are labelled by the same element in Λ.

Note that the previous notion does not treat the silent letter ε in a specific way (like one would do for weak bisimulation): it is considered as a standard letter.

The following proposition follows by definition.

Proposition 3. *Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be two trace-equivalent labelled rooted deterministic digraphs (possibly with ε-labelled edges). Then the trees obtained by unfolding the ε-closure of those two digraphs are the same.*

Let $\mathrm{Graph}_\ell(\mathsf{CPDA}(G))$ be the vertex labelled version of $\mathrm{Graph}(\mathsf{CPDA}(G))$ (with the initial configuration as its root) that is obtained by labelling every vertex with the label of the topmost symbol of the corresponding stack if it belongs to Σ and by $\sharp$ otherwise. We have the following key result (to be proved later).

Table 3: Definition of the transform $\mathsf{CPDA}(G)$.

If u's label is not a variable, the action is just a $push_1^v$, where v is an appropriate child of the node u. Precisely:

(A) If the label is an @ then $\delta(u,\varepsilon) = push_1^{E_0(u)}$.

(S) If the label is a Σ-symbol f then $\delta(u,i) = push_1^{E_i(u)}$, for every $1 \leq i \leq ar(f)$.
Note that if f is nullary, the automaton terminates.

(L) If the label is a lambda then $\delta(u,\varepsilon) = push_1^{E_1(u)}$.

Suppose u is labelled by a variable which is the i-parameter of the lambda node $binder(u)$; and suppose $binder(u)$ is a j-child. Let p be the span of the variable node u.

($\mathbf{V}_1$) If the variable has order $l \geq 1$, then

$$\delta(u,\varepsilon) \quad = \quad \begin{cases} push_{n-l+1}\,;\, pop_1^{p+1}\,;\, push_1^{E_i(top_1),n-l+1} & \text{if } j = 0 \\ push_{n-l+1}\,;\, pop_1^{p}\,;\, collapse\,;\, push_1^{E_i(top_1),n-l+1} & \text{otherwise} \end{cases}$$

where pop_1^p means the operation pop_1 iterated p times, and $push_1^{E_i(top_1),k}$ is defined to be the operation $s \mapsto push_1^{E_i(top_1\, s),k}\, s$.

($\mathbf{V}_0$) Otherwise (*i.e.* the variable has order 0)

$$\delta(u,\varepsilon) \;=\; \begin{cases} pop_1^{p+1}\,;\, push_1^{E_i(top_1)} & \text{if } j = 0 \\ pop_1^{p}\,;\, collapse\,;\, push_1^{E_i(top_1)} & \text{otherwise.} \end{cases}$$

Theorem 5. *For every order-n recursion scheme G,* $\mathrm{Graph}_\ell(\mathsf{CPDA}(G))$ *and* $Tr(G)$ *are trace-equivalent.*

An important consequence of Theorem 5 is the following.

Theorem 6 (Equi-Expressivity 2)**.** *For every order-n recursion scheme G,* $\mathsf{CPDA}(G)$ *(together with the identity labelling function) generates (the same tree as) the value tree* $[\![G]\!]$.

Proof. Take an order-n recursion scheme G. Using Theorem 5, $\mathrm{Graph}_\ell(\mathsf{CPDA}(G))$ and $Tr(G)$ are trace-equivalent. By definition, the tree generated by $\mathsf{CPDA}(G)$ and the identity function, coincides with that obtained by unfolding the ε-closure of $\mathrm{Graph}_\ell(\mathsf{CPDA}(G))$. Hence, using Proposition 3, this tree coincides with that obtained by unfolding the ε-closure of $Tr(G)$, which is $TrTree(G)$. As the latter coincides with the value tree $[\![G]\!]$ (Theorem 4), it concludes the proof. □

6.4 Proof of Theorem 5

We now turn to the technical core of this section. Fix an order-n recursion scheme G. In order to prove that $\mathrm{Graph}_\ell(\mathsf{CPDA}(G))$ and $Tr(G)$ are trace-equivalent, it suffices to establish the following.

Property 7. *Suppose*

$$s_1 \xrightarrow{x_1} s_2 \xrightarrow{x_2} s_3 \xrightarrow{x_3} \cdots \xrightarrow{x_{m-1}} s_m.$$

is a path in $\mathrm{Graph}_\ell(\mathsf{CPDA}(G))$ *starting from the root and*

$$t_1 \xrightarrow{x_1} t_2 \xrightarrow{x_2} t_3 \xrightarrow{x_3} \cdots \xrightarrow{x_{m-1}} t_m$$

is a path in $Tr(G)$ starting from the root (i.e. the trivial traversal ε). Suppose s_m, t_m have the same label from $\Sigma \cup \{\sharp\}$. Then neither path can be extended or both can be extended in the same way:

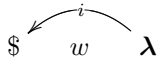
- *by a unique ε-labelled edge (if the final vertices of the paths are labelled by $\sharp$) or*
- *by $ar(f)$ edges with labels in $\{1, \cdots, ar(f)\}$ (if the final vertices of the paths are labelled by $f \in \Sigma$).*

Moreover, if each path is extended by an edge with the same label, the resulting paths end in vertices with the same label.

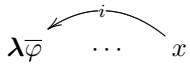
In order to prove the above, we spell out in detail how both paths are related. First, we shall see that, for each $1 \leq i \leq n$, the sequence of node labels corresponding to the top 1-stack, written $\lambda_G(top_2(s_i))$, is the P-view of t_i. *i.e.*

$$\lambda_G(top_2(s_i)) \quad = \quad \ulcorner t_i \urcorner.$$

Secondly we construct a kind of approximant of t_i, written $\widehat{t_i}$, which is obtained from t_i by removing all segments w sandwiched between matching pairs of the shape



where \$ is either an order-1 variable or an @-symbol, and $i \geq 1$, and we do it from right to left. Note that by definition of traversal, the segment w necessarily has the shape



where x is an order-0 variable symbol and $\overline{\varphi}$ is a list of variables in which x occurs. Finally, we remove all pointers from $\widehat{t_i}$. See Example 10 for an illustration of this construction. We then transform each n-stack s_i to a sequence of nodes $\underline{s_i}$, which will be shown to coincide with $\widehat{t_i}$.

Remark 8. *Note that* $\mathsf{CPDA}(G)$ *handles variables of order* 0 *differently from those at higher orders. One could have treated level* 0 *in the same way to obtain correspondence with t (rather than $\widehat{t}$), but this would cost an extra stack level.*

In order to construct the sequence $\underline{s_i}$ from an n-stack s_i, we follow a simple recipe.

1. We "flatten" the n-stack s_i so that it has the form of a well-bracketed sequence such as the following (top of stack is the right-hand end)

$$\texttt{[[[}\cdots\texttt{]}\cdots\texttt{[}\cdots\texttt{]]}\texttt{[[}\cdots\texttt{]]}\cdots\texttt{[[}\cdots\texttt{]}\texttt{[}\cdots\texttt{]]]}$$

2. The target of any pointer to a stack is deemed to be the rightmost symbol representing the stack, *i.e.* it is always an occurrence of `]`.

3. The required subsequence – which we shall write as $\underline{s_i}$ – is obtained by a right-to-left scan of the well-bracketed sequence above according to the following rules.

   - When an occurrence of `]` is encountered, we simply continue the scan without recording `]`.
   - We record any stack symbols that are being scanned.
   - Whenever we encounter the source of a link of order 2 or more, the scan jumps to its target (an occurrence of `]`) without recording any nodes sandwiched in-between. The source of the link is always recorded.
   - The scan ends as soon as some `[` is hit.

   Note that the last condition is necessary to ensure that $\underline{s}$ is suitably defined for every prefix of a reachable stack. This will be important in the proof of Property 8.

Here is a more formal definition.

Definition 5. *Let s be an n-stack. The sequence $\underline{s}$ of stack symbols is defined as follows.*

$$\underline{s} = \begin{cases} \varepsilon & top_2(s) = \texttt{[]}, \\ \underline{(pop_1(s))}\,\lambda_G(u) & top_1(s) = u \text{ and } u \text{ has a 1-link}, \\ \underline{(collapse(s))}\,\lambda_G(u) & top_1(s) = u \text{ and } u \text{ has a } k\text{-link with } k > 1. \end{cases}$$

The next definition relates configurations of $\mathsf{CPDA}(G)$ with traversals over $\lambda(G)$.

Definition 6. *Let G be an order-n recursion scheme, let s be a reachable configuration of $\mathsf{CPDA}(G)$, and let t be a traversal over $\lambda(G)$. We shall say that s **computes** t if and only if the following conditions hold.*

(a) $\lambda_G(top_2(s)) = \ulcorner t \urcorner$.

(b) $\underline{s} = \widehat{t}$.

(c) Suppose $top_2(s) = [v_1, \cdots, v_n]$. Let $v_1', \cdots, v_n'$ be the respective occurrences of $v_1, \cdots, v_n$ in t that contribute to $\ulcorner t \urcorner$. Then $pop_1^{n-i}(s)$ computes $t_{\leq v_i'}$ for every $1 \leq i < n$.

(d) Using the same notation as in (c), suppose v_i has a link to an l-stack σ. Let s_σ be the prefix of s such that σ is its top l-stack, i.e. $s_\sigma = collapse(pop_1^{n-i}(s))$. Then s_σ computes $t_{<v_i'}$.

Note that the definition is not circular, since $t_{\leq v_i'}$ $(1 \leq i < n)$ and $t_{<v_i'}$ $(1 \leq i \leq n)$ are strictly shorter than t. In what follows we shall blur the distinction between v_i and its occurrence v_i', as it will be clear from the context which occurrence is meant. The notion of computing traversals is stable under higher-order pushes, as specified in the next lemma.

Lemma 2. *Let s be an n-stack, $s' = push_k(s)$ and $k \geq 2$. Given an n-stack s'' such that $s < s'' \leq s'$, let $s_{s''}$ be the prefix of s obtained after the same sequence of pop-operations as that used to obtain s'' from s'. Then s'' computes t if and only if $s_{s''}$ computes t for any traversal t.*

Proof. By induction on the length of s''. The base case of s'' is trivial: we have $top_{k+1}(s'') = \bot_k = top_{k+1}(s_{s''})$ and the empty traversal is computed in both cases.

For the inductive step, we need to show that s'' and $s_{s''}$ compute the same traversals. To that end, observe that $top_2(s'') = top_2(s_{s''})$, because – according to the definition of $push_k$ – $top_2(s'')$ is a clone of $top_2(s_{s''})$. Note also that $push_k$ preserves links. Hence, $\underline{s''} = \underline{s_{s''}}$. Thus, conditions (a) and (b) are fulfilled for the same traversal. For (c) and (d), recall that $top_2(s'') = top_2(s_{s''})$ and observe that the requirements regarding computability of traversals by pop_1^{n-i} (for (c)) and ${s''}_\sigma$ (for (d)) are covered by the inductive hypothesis. □

The following implies Property 7.

Property 8. *Suppose*

$$s_1 \xrightarrow{x_1} s_2 \xrightarrow{x_2} s_3 \xrightarrow{x_3} \cdots \xrightarrow{x_{m-1}} s_m$$

is a path in $\mathrm{Graph}_\ell(\mathsf{CPDA}(G))$ *starting from the root and*

$$t_1 \xrightarrow{x_1} t_2 \xrightarrow{x_2} t_3 \xrightarrow{x_3} \cdots \xrightarrow{x_{m-1}} t_m$$

is a path in $Tr(G)$ starting from the root (i.e. the trivial traversal ε). If s_m and t_m are labelled by the same element of $\Sigma \cup \{\sharp\}$, then the following conditons hold.

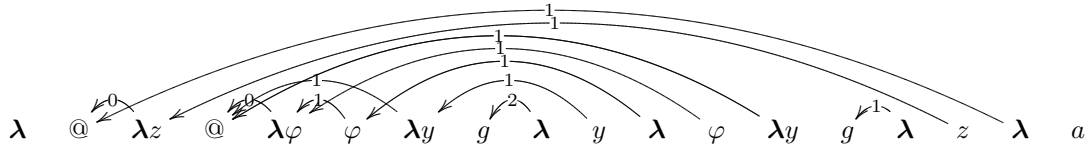
(i) Let $u = top_1(s_m)$. Then u has a link in s_m if and only if it is a j-child ($j > 0$) labelled by a lambda of type A[11] which has order $l \geq 1$. Further, if u has a link, it points to an $(n-l)$-stack.

(ii) s_i computes t_i for all $1 \leq i \leq m$.

Note that (ii) implies that $top_1(s_m)$ is the same as the final vertex in t_m. Consequently, the respective definitions of traversals and $\mathsf{CPDA}(G)$ imply Property 7.

Before going to the proof, we should give some examples that illustrate the relationship between paths in $\mathsf{CPDA}(G)$ and in $Tr(G)$.

Exampl 10. *Take the following traversal over the computation tree of G (recall that variable z has order 0 and variable φ has order 1) in Example 8 (see also Figure 2 for the associated HORS graph):*



In Figure 3 we give the path of the corresponding 2-CPDA that ends in a configuration that computes the above traversal. For ease of reading, in Figure 3 (and later in Figures 6 and 7), instead of stack symbols we shall write their image by Λ_G rather than the exact symbol from V.

[11] We are abusing notation here. Technically, $\boldsymbol{\lambda}\overline{\psi}$ is a terminal symbol from the long transform $\overline{G}$, hence it should have order 0 or 1. However, by the *type* of $\boldsymbol{\lambda}\overline{\psi}$ we mean $(type(\psi_1), \ldots, type(\psi_k), o)$, where $\overline{\psi} = \psi_1 \cdots \psi_k$.

Figure 3: A run of a 2-CPDA

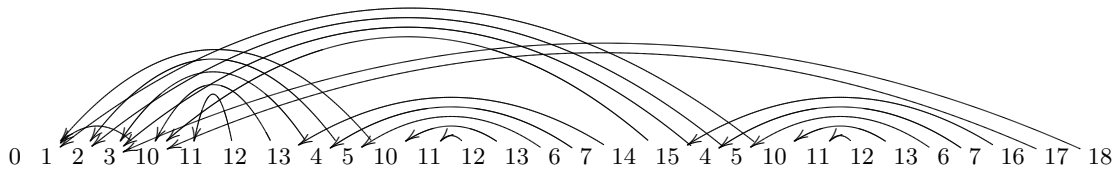


Figure 4: An order-3 traversal

To save space, we only present the interesting configurations in which the top_1-element of the stack is a variable node. In the picture, the top of a stack is at the right-hand end, and links are represented by dotted arrows. Set t to be the prefix of the above traversal that ends in the node labelled by z. We have

$$\widehat{t} = \boldsymbol{\lambda} \quad @ \quad \boldsymbol{\lambda} z \quad @ \quad \boldsymbol{\lambda}\varphi \quad \varphi \quad \boldsymbol{\lambda} \quad \varphi \quad \boldsymbol{\lambda} y \quad g \quad \boldsymbol{\lambda} \quad z$$

which coincides with the 2-stack $\underline{s}$ (s is marked in Figure 3) by following the recipe.

Exampl 11. *Consider the order-3 HORS graph in Figure 5 where variables x_1, x_2, z have order 0, variable φ has order 1 and variable Ψ has order 2. For ease of reference, we give nodes numeric names, which are indicated (within square-brackets) as superscripts. Take the traversal $\mathbf{t}$ in Figure 4.*

We present a run of the 3-CPDA that computes the traversal $\mathbf{t}$ in Figure 6 followed by Figure 7 (for ease of reading, we represent nodes by their labels).

To see the correspondence with the traversal $\mathbf{t}$, note that configurations s_2 and s_3 in Figures 6

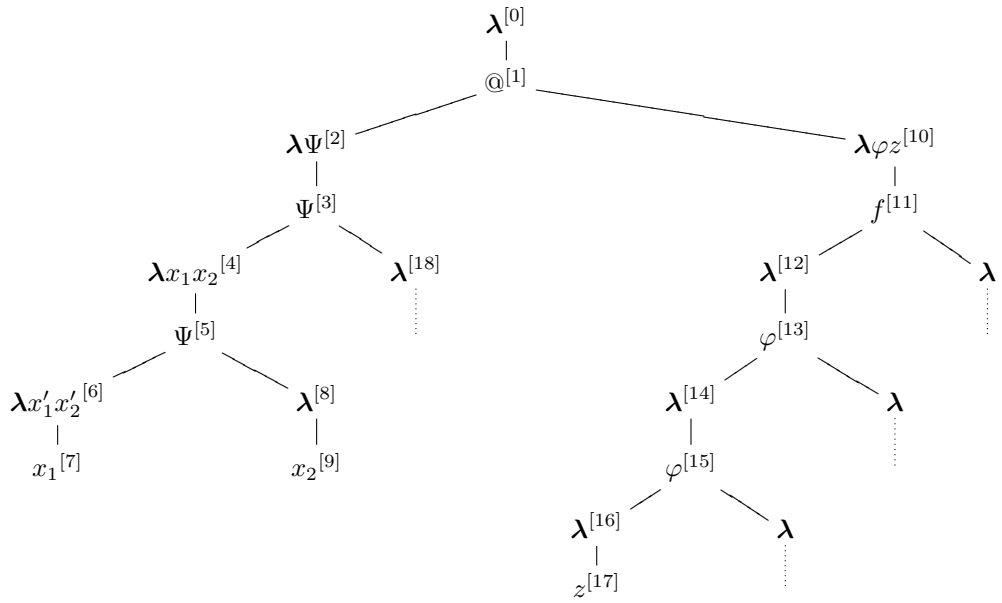


Figure 5: An example of an order-3 HORS graph.

and 7 respectively have the same top_1-element which is node 7 (labelled by x_1). They correspond respectively to the two prefixes of $\mathbf{t}$ *that end in node 7.*

The traversal t corresponding to s_3 is the prefix of $\mathbf{t}$ *that ends in the later occurrence of 7; we have*

$$\widehat{t} \quad = \quad \boldsymbol{\lambda} \;\; @ \;\; \boldsymbol{\lambda}\Psi \;\; \Psi \;\; \boldsymbol{\lambda}\varphi z \;\; f \;\; \boldsymbol{\lambda} \;\; \varphi \;\; \boldsymbol{\lambda} \;\; \varphi \;\; \boldsymbol{\lambda}x_1x_2 \;\; \Psi \;\; \boldsymbol{\lambda}\varphi z \;\; f \;\; \boldsymbol{\lambda} \;\; \varphi \;\; \boldsymbol{\lambda}x_1'x_2' \;\; x_1$$

The reader might wish to check that $\underline{s_3} = \widehat{t}$. (Note that the justification pointers are uniquely reconstructible from the underlying sequence of nodes and their respective labels.)

The rest of the section is devoted to the proof of Property 8. The proof is by induction on m. Clearly the above assertions are valid when $m = 1$. For the inductive case, we assume that the property holds for some $m \geq 1$.

We shall do so by a case analysis of the label of $top_1(s_m) = u$.

First suppose u's label is not a variable. Then $s_{m+1} = push_1^v(s_m)$, for an appropriate node. In particular, no new link is created.

For (i), observe that, because u's label is not a variable, it follows from the definition of $\mathsf{CPDA}(G)$ that, if v was a j-child labelled by a lambda of type A, then u would have to be labelled by a Σ-symbol and, thus, the order of A would be 0.

For (ii), $t_m \xrightarrow{x_m} t_{m+1}$ implies that $t_{m+1} = t_m v$, where v has a pointer to a suitable node (there is only one way in which a pointer from v can be inserted so as to make t_{m+1} into a traversal). We shall show that s_{m+1} computes t_{m+1}.

For (a), we need to check that $\lambda_G(top_2(s_{m+1})) = \ulcorner t_{m+1} \urcorner$. We have $\lambda_G(top_2(s_{m+1})) = \lambda_G(top_2(s_m))v$ and, in all three cases corresponding to the rules (**A**), (**S**), (**L**), $\ulcorner t_{m+1} \urcorner = \ulcorner t_m \urcorner v$ holds. Thus, by induction hypothesis, we get $\lambda_G(top_2(s_{m+1})) = \ulcorner t_{m+1} \urcorner$. For (b), we note that

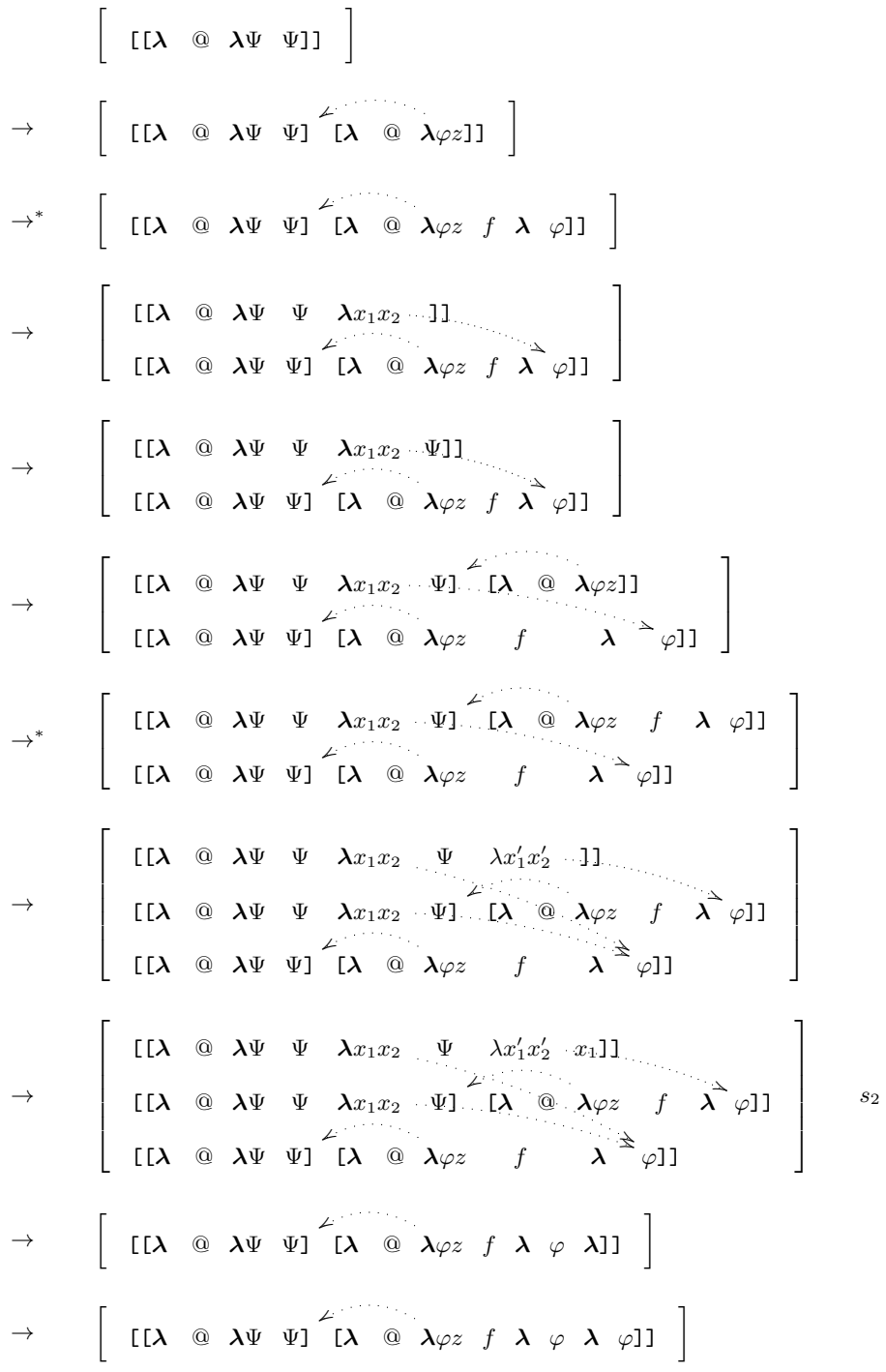


Figure 6: A run of a 3-CPDA (Part 1 of 2).

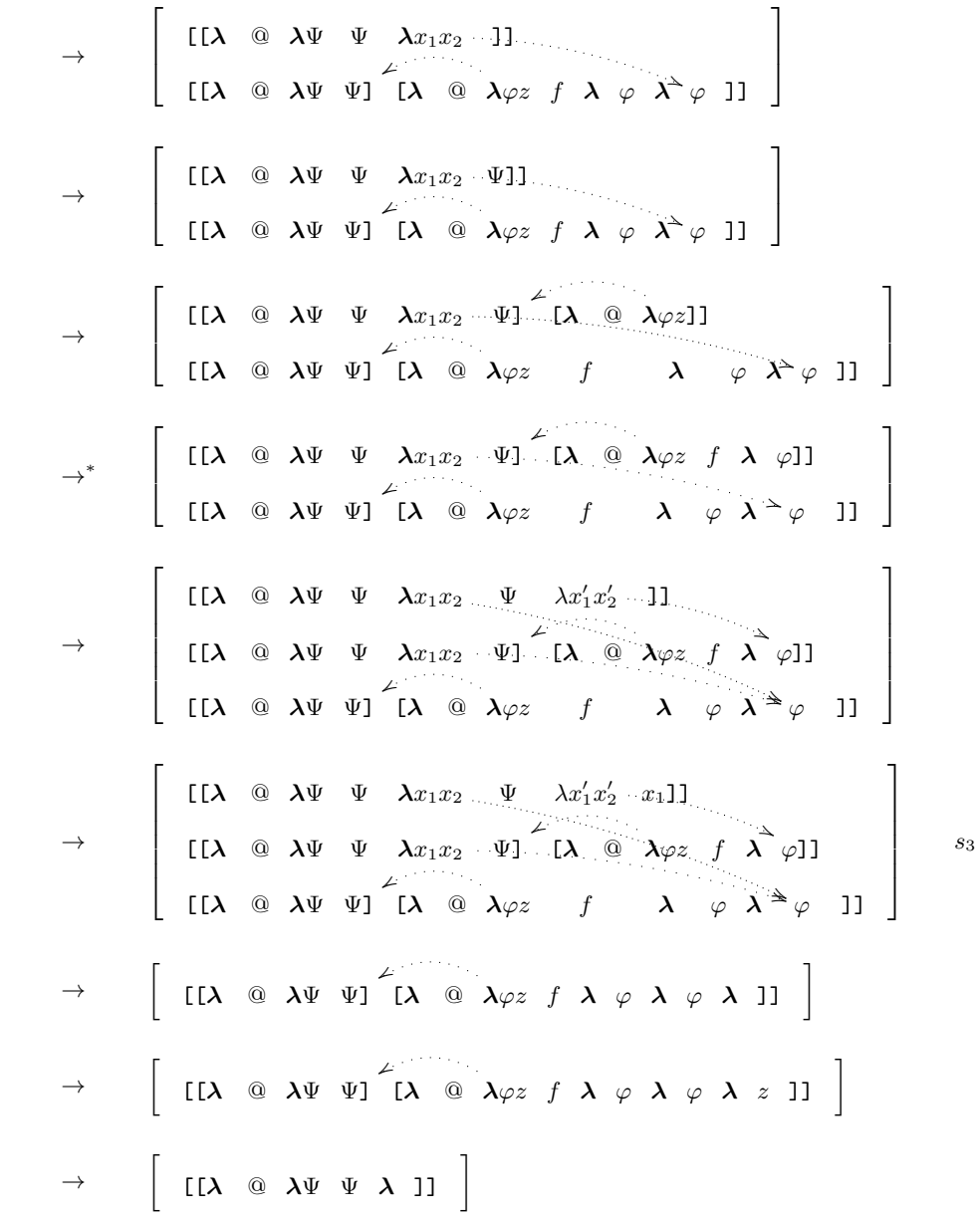


Figure 7: A run of a 3-CPDA (Part 2 of 2).

$\underline{s_{m+1}} = \underline{s_m}v$ and $\widehat{t_{m+1}} = \widehat{t_m}v$. So, by induction hypothesis, $\underline{s_{m+1}} = \widehat{t_{m+1}}$. Condition (c) follows immediately from the induction hypothesis and, because no new links have been created, so does (d).

Next suppose u's label is an order-l variable, which is the i-parameter of $binder(u)$ (note that then we have $i \geq 1$) and suppose $binder(u)$ is a j-child. Then $s_{m+1} = \delta(u)(s_m)$ where $\delta(u)$ is given in Definition 4. There are four cases; in the following we shall use the notations from Definition 4. In order to simplify the notations, we should refer to s_m (*resp.* t_m) as s (*resp.* t) and to s_{m+1} (*resp.* t_{m+1}) as s' (*resp.* t').

1. Case $l \geq 1$ ***and*** $j = 0$**.** Let φ_i be the order-l variable labelling u.

By the induction hypothesis of (ii), u must be the last node of t. It then follows from the definition of a traversal (and from $binder(u)$ being a 0-child) that t has the following shape:

$$t \quad = \quad \cdots \; \overset{0}{\underset{@}{u_0}} \; \overset{i}{\underset{\boldsymbol{\lambda}\overline{\varphi}}{u_1}} \; \cdots \; \underset{\varphi_i}{u}$$

(in the figure, the label of a node is the symbol just below it). Since the P-view of a traversal satisfies the pointer condition and is a path in the HORS graph (Proposition 2), $\ulcorner t \urcorner$ has the shape $\cdots u_0 \underbrace{u_1 \cdots u}_{\theta}$ and the segment θ has length $p+1$, where p is the span of the variable node u. Indeed u belongs to $\ulcorner t \urcorner$ and so does u_1 thanks to the pointer condition; the length of θ comes from $\ulcorner t \urcorner$ being a path.

Consider the operation $\delta(u) = push_{n-l+1}; pop_1^{p+1}; push_1^{E_i(top_1),n-l+1}$. By the induction hypothesis of (ii), the top 1-stack of s — call it σ — is the P-view of t. Since the top 1-stack of $push_{n-l+1}s$ is a copy of σ, applying pop_1^{p+1} to $push_{n-l+1}(s)$ returns a stack that has the @-labelled node u_0 as the top_1-element. The node that is pushed onto the top of the stack at this point is the i-child of u_0, which we call v. Further, it has a link to the top $(n-l)$-stack of the prefix s of s', hence $collapse(s') = s$.

It follows from the structure of $\lambda(G)$ that v must be labelled by $\boldsymbol{\lambda}\overline{\psi}$ (say) of the same type as the label φ_i of u, *i.e.* its type is also of order $l \geq 1$. Thus, since $i \geq 1$, (i) follows as required.

For (ii), observe that $t' = tv$, where v has a pointer (labelled by i) to the occurrence of u_0 indicated in the figure above. Also, we have $t \xrightarrow{\varepsilon} t'$. We shall show that s' computes t'.

(a) We need to show $\lambda_G(top_2(s')) = \ulcorner t' \urcorner$. By definition of s', we have $\lambda_G(top_2(s')) = \lambda_G(top_2(s)_{\leq u_0})v$, *i.e.* $top_2(s')$ is the prefix of the 1-stack $top_2(s)$ — regarded as a sequence — up to and including the occurrence of u_0 described above, extended by v. By induction hypothesis (a), we have $\lambda_G(top_2(s)) = \ulcorner t \urcorner$. Thus

$$\lambda_G(top_2(s')) = \ulcorner t \urcorner_{\leq u_0} v = \ulcorner t_{\leq u_0} \urcorner v = \ulcorner t' \urcorner$$

as required (the second equation holds because u_0 appears in $\ulcorner t \urcorner$).

(b) We have $\underline{s'} = \underline{s}v$ (indeed v has a link and *collapse* $s' = s$) and $\widehat{t'} = \widehat{t}v$. Since $\underline{s} = \widehat{t}$ by induction hypothesis (b), we have $\underline{s'} = \widehat{t'}$.

(c) Because the top 1-stack of s' is (a copy of) a prefix of $top_2(s)$ extended with v, we can simply appeal to the induction hypothesis (ii) – namely, part (c) and the fact that s computes t.

(d) For the same reason as above, (d) holds for all links in $top_2(s')$ except (possibly) the single new link. Let σ' be the $(n-l)$-stack pointed at from v. Then we have $s'_{\sigma'} = s$. Because $t = t'_{<v}$ and s computes t, (d) also holds for the new link.

2. Case $l \geq 1$ ***and*** $j > 0$. Suppose the label of u is the order-l variable φ_i, which is the ith item of the list $\overline{\varphi}$.

By induction hypothesis (ii) and definition of a traversal, t has the following shape:

$$\begin{array}{ccccccccc} t & = & \cdots & u_0 & \overset{\overset{i}{\curvearrowleft}}{u_1} & \cdots & u \\ & & & \psi & \boldsymbol{\lambda}\overline{\varphi} & & \varphi_i \end{array}$$

Further, the variable ψ has the same type as $\boldsymbol{\lambda}\overline{\varphi}$, which (say) is of order l'. It follows that $l' > l$ and, consequently, $l' \geq 1$. By induction hypothesis (i) and (ii), s has the following shape

$$s = \underbrace{[\cdots\cdots\cdots[\cdots\ \sigma\ \cdots\ \overset{\overset{\curvearrowleft}{}}{[\cdots u_1 \cdots u]}\ ]\cdots]}_{\text{top } (n-l)\text{-stack of } s,\ \textit{i.e. } w}$$
$$\text{top 1-stack of } s$$

wherein u_1 has a link to some $(n-l')$-stack σ. Since $l' > l$, the $(n-l')$-stack σ is embedded in the top $(n-l)$-stack of s, as indicated by the figure above. Note that, by induction hypothesis (ii (d)), s_σ computes $t_{<u_1}$. In particular the top_1-element of σ must be u_0.

Now, to see the structure of s', consider the operation $\delta(u)$. Let $w = top_{n-l+1}(s)$. The operation $push_{n-l+1}(s)$ pushes a copy of w on top of s. The rest of $\delta(u)$, namely,

$$pop_1^p; collapse; push_1^{E_i(top_1),n-l+1},$$

affects only the top (duplicate) copy of w. Applying pop_1^p to $push_{n-l+1}(s)$ returns a stack that has u_1 as the top_1-element; the *collapse*-operation then reduces it to a stack that has a copy σ' (say) of σ as its top $(n-l')$-stack, *i.e.* its top_1-element is u_0. The node that is $push_1$ed onto the top of the stack at the end of the $\delta(u)$-operation (to yield s') is the i-child of u_0, which we shall call v. Observe that the structure of $\lambda(G)$ implies that v must then be labelled by $\boldsymbol{\lambda}\overline{\chi}$ (say) whose type is the same as that of φ_i, *i.e.* its order is l. Since v is linked to the $(n-l)$-stack w, (i) is satisfied.

For (ii), observe that $t' = tv$, where v has an i-pointer to the distinguished occurrence of u_0. Note that we then have $t \xrightarrow{\varepsilon} t'$. We need to show that s' computes t'.

(a) Observe that $\ulcorner t' \urcorner = \ulcorner t_{<u_0} \urcorner u_0 v = \ulcorner t_{<u_1} \urcorner v$. Since s_σ computes $t_{<u_1}$ (by induction hypothesis (ii (d))), so does $s'_{\sigma'}$ by Lemma 2. Hence, $\lambda_G(top_2(s')) = \lambda_G(top_2(\sigma'))v = \ulcorner t_{<u_1} \urcorner v = \ulcorner t' \urcorner$.

(b) Observe that $\underline{s'} = \underline{s}v$ (indeed v has a link and $collapse(s') = s$) and $\widehat{t'} = \widehat{t}v$. Thus, by induction hypothesis, $\underline{s'} = \widehat{t'}$.

(c) Since $s'_{\sigma'}$ computes $t_{<u_1}$ and $top_2(s')$ is a copy of $top_2(s'_{\sigma'})$ augmented with v, (c) holds.

(d) We only need to verify (d) for the new link (all other links satisfy (d) because $s'_{\sigma'}$ computes $t_{<u_1}$). Recall that v points at the stack w. Since $s'_w = s$ and $t'_{<v} = t$, (d) holds because s computes t.

3. Case $l = 0$ ***and*** $j = 0$**.** Suppose u's label is the order-0 variable x.

By induction hypothesis (ii) and the definition of a traversal (and from $binder(u)$ being a 0-child), t must have the following shape:

$$\begin{array}{ccc} t & = & \cdots \overset{0}{\underset{\swarrow}{\curvearrowleft}} u_0 \quad u_1 \overset{i}{\underset{\swarrow}{\curvearrowleft}} \cdots \quad u \\ & & @ \quad \boldsymbol{\lambda}\overline{\varphi} \qquad\quad x \end{array}$$

As in **1.**, $\ulcorner t \urcorner$ has the shape $\cdots u_0 \underbrace{u_1 \cdots u}_{\theta}$ and the segment θ has length $p+1$, where p is the span of the variable node u.

Consider the operation $\delta(u) = pop_1^{p+1}; push_1^{E_i(top_1)}$. Applying pop_1^{p+1} to s returns a stack that has the @-labelled node u_0 as the top_1-element. The node that is $push_1$ed onto the top of the stack at this point is the i-child of u_0, which we call v. It follows from the structure of $\lambda(G)$ that v must be labelled by $\boldsymbol{\lambda}$, *i.e.* its type has order 0. Thus, since v has no link, (i) follows as required.

For (ii), note that $t' = tv$, where v has a pointer (labelled by i) to the occurrence of u_0 indicated above. We have $t \xrightarrow{\varepsilon} t'$. We shall show that s' computes t'.

(a) We need to show $\lambda_G(top_2(s')) = \ulcorner t' \urcorner$. By definition of s', we have $top_2(s') = (top_2(s))_{\leq u_0} v$. By induction hypothesis (ii), we have $\lambda_G(top_2(s)) = \ulcorner t \urcorner$. Thus

$$\lambda_G(top_2(s')) = \ulcorner t \urcorner_{\leq u_0} v = \ulcorner t_{\leq u_0} \urcorner v = \ulcorner t' \urcorner$$

as required. The second equation follows from the definition of $\ulcorner\urcorner$ and the fact that u_0 appears in $\ulcorner t \urcorner$.

(b) We have $\underline{s'} = \underline{(pop_1^{p+1}(s))}\, v$ and $\widehat{t'} = \widehat{t_{\leq u_0}} v$. By induction hypothesis (ii (c)), $pop_1^{p+1}(s)$ computes $t_{\leq u_0}$, in particular $\underline{pop_1^{p+1}(s)} = \widehat{t_{\leq u_0}}$. Thus, (b) holds.

(c) We simply appeal to the induction hypothesis (ii). As before, we need (c) and the fact that s computes t.

(d) Note that no new links have been created in this case, so it suffices to appeal to the induction hypothesis (ii (d)).

4. Case $l = 0$ ***and*** $j > 0$**.** Suppose the label of u is the order-0 variable x, which is the ith item of the list $\overline{\varphi}$.

By induction hypothesis (ii) and definition of a traversal, t has the following shape:

$$\begin{array}{ccc} t & = & \cdots \quad u_0 \quad \overset{i}{\underset{\swarrow}{\curvearrowleft}} u_1 \quad \cdots \quad u \\ & & \psi \quad \boldsymbol{\lambda}\overline{\varphi} \qquad\quad x \end{array}$$

Further, the variable ψ has the same type as $\boldsymbol{\lambda}\overline{\varphi}$, which (say) is of order l'. It follows that $l' > l$. By induction hypothesis (i) and (ii), s has the following shape

$$s \quad = \quad [\cdots\cdots \sigma \cdots \underbrace{[\cdots u_1 \cdots u]}_{\text{top 1-stack}} \cdots]$$

wherein u_1 has a link to some $(n-l')$-stack σ. Note that, by induction hypothesis (ii (d)), s_σ computes $t_{<u_1}$. In particular the top_1-element of σ must be u_0.

Now, to understand what s' looks like, consider the operation $\delta(u) = pop_1^p; collapse; push_1^{E_i(top_1)}$. Applying pop_1^p to s returns a stack that has u_1 as the top_1-element; the *collapse*-operation then reduces it to a stack that has σ as its top $(n-l')$-stack, *i.e.* its top_1-element is u_0. The node that is then $push_1$ed onto the top of the stack at the end of the $\delta(u)$-operation (to yield s') is the i-child of u_0, which we shall call v. Observe that the structure of $\lambda(G)$ implies that v must then be labelled by $\boldsymbol{\lambda}$. Since v does not have a link, (i) is satisfied.

For (ii) let $t' = tv$, where v has an i-pointer to the distinguished occurrence of u_0. Then t' is a traversal such that $t \xrightarrow{\varepsilon} t'$. We need to show that s' computes t'.

(a) Observe that $\ulcorner t' \urcorner = \ulcorner t_{<u_1} \urcorner v$. Since s_σ computes $t_{<u_1}$, we have $top_2(s') = \ulcorner t_{<u_1} \urcorner v = \ulcorner t' \urcorner$.

(b) Observe that $\underline{s'} = \underline{s_\sigma} v$ and $\widehat{t'} = \widehat{t_{<u_1}} v$. Again, since s_σ computes $t_{<u_1}$, we have $\underline{s_\sigma} = \widehat{t_{<u_1}}$ and (b) follows.

(c) Because s_σ computes $t_{<u_1}$, condition (c) holds.

(d) Again, it suffices to appeal to the fact that s_σ computes $t_{<u_1}$, because no new links have been created.

7 Conclusion

In this paper, we introduced *collapsible pushdown automata* and proved that they are equi-expressive with (general) recursion schemes for generating trees. This is the first automata-theoretic characterisation of higher-order recursions schemes in full generality.

Due to its length, we decided to restrict this paper to the full proof of the Equi-Expressivity Theorem (that was originally stated in [34]). In particular, we had to postpone those questions coming from logic and games. We now briefly discuss the main results in this field as well as other consequences of the Equi-Expressivity.

The Equi-Expressivity Theorem is significant because it acts as a bridge, enabling inter-translation between model-checking problems about trees generated by recursion scheme and model-checking problems/solvability of games on collapsible pushdown graphs. Indeed, consider a μ-calculus formula φ and a transition graph $\mathrm{Graph}(\mathcal{A})$ of a CPDA. Deciding whether φ holds in some vertex v of the graph is equivalent to decide whether the same formula φ is true at the root of the tree obtained by unfolding $\mathrm{Graph}(\mathcal{A})$ from v. As this tree can be obtained as the value tree of some scheme G, the original question is reduced to decide validity of a μ-calculus formula at the root of $[\![G]\!]$. Of course this chain of reductions works in the other direction as well. In particular, the results of [53] imply that μ-calculus model-checking is decidable for transition graphs of CPDA.

As μ-calculus model-checking for transition graphs of CPDA is equivalent to solving parity games played on transition graphs of CPDA, it was a natural question to study these games in order to transfer back decidability results to recursion schemes. We showed in [34] that those games are decidable (actually they are n-ExpTime complete for order-n CPDA transition graphs), hence leading to an alternative proof of [53] for the decidability of MSO/μ-calculus model-checking for trees generated by recursion schemes.

Later, by carefully studying these games, Broadbent, Carayol, Ong and Serre showed in [9] that the winning regions of these games admit a finite representation that can later be used (in a non-trivial way and strongly relying on the Equi-Expressivity Theorem) to prove that recursion

schemes are *constructively reflective* with respect to μ-calculus and MSO[12]. This result was later subsumed by a result of Carayol and Serre showing that recursion schemes enjoy the effective MSO selection property [15]. The main tools to prove this result are the equi-expressivity theorem and a careful analysis of the winning strategies of parity games played on transition graph of CPDA.

An important problem on recursion schemes was known as the safe/unsafe conjecture. It asked whether there exists for all schemes, another scheme having the same value tree and verifying the safety constraint. The Equi-Expressivity Theorem permits to rephrase this question as whether CPDA are equi-expressive with higher-order PDA *for generating trees*. The conjecture was that the former are strictly more expressive. A first step in this direction was obtained by Parys who proved that the *Urzyczyn language* (which is a language of finite words) is definable by a 2-CPDA but not by any deterministic 2-PDA [57]. The proof was obtained by reasoning about accepting runs of 2-PDA, and thus the result on schemes was made possible thanks to the Equi-Expressivity Theorem. Parys recently extended this result by showing that the Urzyczyn language cannot be defined by any deterministic n-PDA (for any n) [58].

In [38] Karzow and Parys gave a pumping lemma for collapsible pushdown automata that, thanks to the Equi-Expressivity Theorem, establishes the strictness of the hierarchy $(RecTree_n\Sigma)_n$ of trees generated by n-CPDA. More precisely, they gave, for every $n \geq 0$, a tree generated by an order-$(n+1)$ (safe) scheme that no order-n scheme can generate.

The Equi-Expressivity Theorem also opened a new avenue of model-checking algorithms for recursion schemes. The saturation technique – underlying the Moped tool [65, 71, 70] for reachability analysis of PDA – was extended in [11] to CPDA, which led to the implementation of the C-SHORe tool [12] for recursion scheme model-checking. This automata-theoretic approach contrasted with existing tools (TRecS [42], GTRecS(2) [44, 45], and TravMC [51]) that are based on intersection type checking. This work also inspired the HorSat tool, which transferred the saturation method to the intersection type setting [10]. For completeness, we also mention the recent Preface tool [60], a fast counter-example abstraction-refinement based model checker for recursion.

Another advantage of the automata-theoretic view on recursion schemes is the transferral of techniques for reasoning about concurrent systems. The search for concurrent extensions of pushdown automata with decidable model-checking properties has been well-studied, and it has been shown that a number of these extensions can be generalised to CPDA [33].

We conclude with a brief discussion of further directions:

1. Is there an *à la* Caucal definition for the ε-closure of CPDA graphs? As trees generated by n-CPDA are exactly those obtained by unravelling and unfolding an n-CPDA graph, is there a class of transformations $\mathcal{T}$ from trees to graphs such that every $(n+1)$-CPDA graph is obtained by applying a $\mathcal{T}$-transformation to some tree generated by an n-CPDA. Note that a $\mathcal{T}$-transformation may in general not preserve MSO decidability (as n-CPDA graphs have undecidable MSO theory [34]), but should preserve modal μ-calculus decidability of trees generated by n-CPDA.

[12]Let $\mathcal{R}$ be a class of generators of node-labelled infinite trees, and $\mathcal{L}$ be a logical language for describing correctness properties of these trees. Given $R \in \mathcal{R}$ and $\varphi \in \mathcal{L}$, we say that R_φ is a *φ-reflection* of R just if

- R and R_φ generate the same underlying tree, and
- suppose a node u of the tree $[\![R]\!]$ generated by R has label f, then the label of the node u of $[\![R_\varphi]\!]$ is $\underline{f}$ if u in $[\![R]\!]$ satisfies φ; it is f otherwise.

Thus if $[\![R]\!]$ is the computation tree of a program R, we may regard R_φ as a transform of R that can internally observe its behaviour against a specification φ. We say that $\mathcal{R}$ is (constructively) *reflective* w.r.t. $\mathcal{L}$ just if there is an algorithm that transforms a given pair (R, φ) to R_φ.

2. The deepest open problem is without any doubt the equivalence problem for higher-order recursion schemes (*i.e.* given two schemes, decide whether they have the same value tree). The Equi-Expressivity Theorem implies that the equivalence problem for schemes is interreducible to the problem of decidability of language equivalence between deterministic CPDA (as words acceptors). Of course this problem is extremely hard, as it would generalise the DPDA equivalence decidability result of Sénizergues [66, 67].

Acknowledgments We would like to warmly thank the reviewers for their numerous highly valuable comments. This work was supported by the Engineering and Physical Sciences Research Council [EP/C539753/1, EP/K009907/1, and EP/M023974/1].

References

[1] Klaus Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. In *Proceeding of Computer Science Logic (CSL 2006), 20th Annual Conference of the EACSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 104–118. Springer-Verlag, 2006.

[2] Klaus Aehlig, Jolie de Miranda, and C.-H. Luke Ong. Safety is not a restriction at level 2 for string languages. Technical Report PRG-RR-04-023, Oxford University Computing Laboratory, 2004.

[3] Klaus Aehlig, Jolie de Miranda, and C.-H. Luke Ong. Safety is not a restriction at level 2 for string languages. In *Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS 2005)*, volume 3411 of *Lecture Notes in Computer Science*, pages 490–501. Springer-Verlag, 2005.

[4] André Arnold and Damian Niwiński. *Rudiments of μ-Calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2001.

[5] Vince Bárány, Erich Grädel, and Sasha Rubin. *Automata-Based Presentations of Infinite Structures*, volume 379 of *London Mathematical Society Lecture Notes Series*, pages 1–76. Cambridge University Press, 2011.

[6] William Blum. Type homogeneity is not a restriction for safe recursion schemes. *CoRR*, abs/1701.02118, 2017.

[7] William Blum and C.-H. Luke Ong. The safe lambda calculus. *Logical Methods in Computer Science*, 5(1), 2009.

[8] Christopher Broadbent. *On Collapsible Pushdown Automata, their Graphs and the Power of Links*. PhD thesis, University of Oxford, 2011.

[9] Christopher Broadbent, Arnaud Carayol, C.-H. Luke Ong, and Olivier Serre. Recursion schemes and logical reflexion. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science (LICS 2010)*, pages 120–129. IEEE Computer Society, 2010.

[10] Christopher Broadbent and Naoki Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *Proceeding of Computer Science Logic (CSL 2013), 27th Annual Conference of the EACSL*, volume 23 of *LIPIcs*, pages 129–148. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.

[11] Christopher H. Broadbent, Arnaud Carayol, Matthew Hague, and Olivier Serre. A saturation method for collapsible pushdown systems. In *Proceedings 39th International Conference on Automata, Languages, and Programming (ICALP 2012)*, volume 7392 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, 2012.

[12] Christopher H. Broadbent, Arnaud Carayol, Matthew Hague, and Olivier Serre. C-SHORe: a collapsible approach to higher-order verification. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP 2013)*, pages 13–24. ACM, 2013.

[13] Thierry Cachat. Higher order pushdown automata, the Caucal hierarchy of graphs and parity games. In *Proceedings 30th International Conference on Automata, Languages, and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 556–569. Springer-Verlag, 2003.

[14] Arnaud Carayol, Antoine Meyer, Matthew Hague, C.-H. Luke Ong, and Olivier Serre. Winning regions of higher-order pushdown games. In *Proceedings of the 23th Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 193–204. IEEE Computer Society, 2008.

[15] Arnaud Carayol and Olivier Serre. Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)*, pages 165–174. IEEE Computer Society, 2012.

[16] Arnaud Carayol and Stefan Wöhrle. The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *Proceedings of 23rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS 2003)*, volume 2914 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 2003.

[17] Didier Caucal. On infinite terms having a decidable monadic theory. In *Proceedings 27th Symposium, Mathematical Foundations of Computer Science (MFCS 2002)*, volume 2420 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, 2002.

[18] Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Trans. Amer. Math. Soc.*, 39(3):472–472, mar 1936.

[19] Bruno Courcelle. A representation of trees by languages I. *Theoretical Computer Science*, 6:255–279, 1978.

[20] Bruno Courcelle. A representation of trees by languages II. *Theoretical Computer Science*, 7:25–55, 1978.

[21] Bruno Courcelle. The monadic second-order logic of graphs IX: Machines and their behaviours. *Theoretical Computer Science*, 151:125–162, 1995.

[22] Bruno Courcelle and Maurice Nivat. The algebraic semantics of recursive program schemes. In *Proceedings 7th Symposium, Mathematical Foundations of Computer Science (MFCS 1978)*, volume 64 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1978.

[23] Werner Damm. Higher type program schemes and their tree languages. In *Theoretical Computer Science, 3rd GI-Conference*, volume 48 of *Lecture Notes in Computer Science*, pages 51–72. Springer-Verlag, 1977.

[24] Werner Damm. Languages defined by higher type program schemes. In *Proceedings 4th International Conference on Automata, Languages, and Programming (ICALP 1977)*, volume 52 of *Lecture Notes in Computer Science*, pages 164–179. Springer-Verlag, 1977.

[25] Werner Damm. The IO- and OI-hierarchies. *Theoretical Computer Science*, 20:95–207, 1982.

[26] Werner Damm and Andreas Goerdt. An automata-theoretical characterization of the oi-hierarchy. *Information and Computation*, 71:1–32, 1986.

[27] Jolie de Miranda. *Structures Generated by Higher-Order Grammars and the Safety Constraint*. PhD thesis, University of Oxford, 2006.

[28] Joost Engelfriet and Erik Meineche Schmidt. IO and OI. I. *Journal of Computer and System Science*, 15(3):328–353, 1977.

[29] Joost Engelfriet and Erik Meineche Schmidt. IO and OI. II. *Journal of Computer and System Science*, 16(1):67–99, 1978.

[30] Jörg Flum, Erich Grädel, and Thomas Wilke. *Logic and Automata: History and Perspectives*. Amsterdam University Press, 2007.

[31] Stephen Garland and David Luckham. Program schemes, recursion schemes and formal languages. *Journal of Computer and System Science*, 7(2):119–160, 1973.

[32] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume

2500 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[33] Matthew Hague. Saturation of concurrent collapsible pushdown systems. In *Proceedings of the 33rd International Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS 2013)*, volume 24 of *LIPIcs*, pages 313–325. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.

[34] Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of the 23th Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 452–461. IEEE Computer Society, 2008.

[35] J. Martin E. Hyland and C.-H. Luke Ong. On Full Abstraction for PCF: I. Models, Observables and the Full Abstraction Problem, II. Dialogue Games and Innocent Strategies, III. A Fully Abstract and Universal Game Model. *Information and Computation*, 163:285–408, 2000.

[36] Klaus Indermark. Schemes with recursion on higher types. In *Proceedings 5th Symposium, Mathematical Foundations of Computer Science (MFCS 1976)*, volume 45 of *Lecture Notes in Computer Science*, pages 352–358. Springer-Verlag, 1976.

[37] Alexander Kartzow. Collapsible pushdown graphs of level 2 are tree-automatic. In *Proceedings 26th Symposium on Theoretical Aspects of Computer Science (STACS 2010)*, volume 5 of *LIPIcs*, pages 501–512. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[38] Alexander Kartzow and Pawel Parys. Strictness of the collapsible pushdown hierarchy. In *Proceedings 37th Symposium, Mathematical Foundations of Computer Science (MFCS 2012)*, volume 7464 of *Lecture Notes in Computer Science*, pages 566–577. Springer-Verlag, 2012.

[39] Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In *Proceedings of Typed Lambda Calculi and Applications, 5th International Conference (TLCA 2001)*, volume 2044 of *Lecture Notes in Computer Science*, pages 253–267. Springer-Verlag, 2001.

[40] Teodor Knapik, Damian Niwiński, and Paweł Urzyczyn. Higher-order pushdown trees are easy. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS 2002)*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer-Verlag, 2002.

[41] Teodor Knapik, Damian Niwiński, Pawel Urzyczyn, and Igor Walukiewicz. Unsafe grammars and panic automata. In *Proceedings 32nd International Conference on Automata, Languages, and Programming (ICALP 2005)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1450–1461. Springer-Verlag, 2005.

[42] Naoki Kobayashi. Model-checking higher-order functions. In *Proceedings of the 11th International ACM SIGPLAN-SIGACT Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, pages 25–36. ACM, 2009.

[43] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 416–428. ACM, 2009.

[44] Naoki Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS 2011)*, volume 6604 of *Lecture Notes in Computer Science*, pages 260–274. Springer-Verlag, 2011.

[45] Naoki 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.

[46] Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science (LICS 2009)*, pages 179–188. IEEE Computer Society, 2009.

[47] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[48] A. N. Maslov. The hierarchy of indexed languages of an arbitrary level. *Soviet Mathematics*

Doklady, 15:1170–1174, 1974.

[49] A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 12:38–43, 1976.

[50] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[51] Robin P. Neatherway, Steven J. Ramsay, and C.-H. Luke Ong. A traversal-based algorithm for higher-order model checking. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pages 353–364. ACM, 2012.

[52] Maurice Nivat. On the interpretation of recursive program schemes. In *Symposia Mathematica*, 1972.

[53] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS 2006)*, pages 81–90. IEEE Computer Society, 2006.

[54] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. Preprint, 2006.

[55] C.-H. Luke Ong. Higher-order model checking: An overview. In *Proceedings of the 30th Annual IEEE Symposium on Logic in Computer Science (LICS 2015)*, pages 1–15. IEEE Computer Society, 2015.

[56] C.-H. Luke Ong. Normalisation by traversals. *CoRR*, abs/1511.02629, 2015.

[57] Paweł Parys. Collapse operation increases expressive power of deterministic higher order pushdown automata. In *Proceedings 28th Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, volume 9 of *LIPIcs*, pages 603–614. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[58] Paweł Parys. On the significance of the collapse operation. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)*, pages 521–530. IEEE Computer Society, 2012.

[59] Michael Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[60] Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. A type-directed abstraction refinement approach to higher-order model checking. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*, pages 61–72. ACM, 2014.

[61] Sylvain Salvati and Igor Walukiewicz. Krivine machines and higher-order schemes. In *Proceedings 38th International Conference on Automata, Languages, and Programming (ICALP 2011)*, volume 6756 of *Lecture Notes in Computer Science*, pages 162–173. Springer-Verlag, 2011.

[62] Sylvain Salvati and Igor Walukiewicz. Recursive schemes, Krivine machines, and collapsible pushdown automata. In *Proceedings of Reachability Problems - 6th International Workshop (RP 2012)*, volume 7550 of *Lecture Notes in Computer Science*, pages 6–20. Springer-Verlag, 2012.

[63] Sylvain Salvati and Igor Walukiewicz. Krivine machines and higher-order schemes. *Information and Computation*, 239:340–355, 2014.

[64] Sylvain Salvati and Igor Walukiewicz. Simply typed fixpoint calculus and collapsible pushdown automata. *Mathematical Structures in Computer Science*, 26(7):1304–1350, 2015.

[65] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical University of Munich, 2002.

[66] Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *Proceedings 24th International Conference on Automata, Languages, and Programming (ICALP 1997)*, volume 1256 of *Lecture Notes in Computer Science*, pages 671–681. Springer-Verlag, 1997.

[67] Géraud Sénizergues. L(A)=L(B)? a simplified decidability proof. *Theoretical Computer Science*, 281(1-2):555–608, 2002.

[68] Colin Stirling. Higher-order matching and games. In *Proceedings of Computer Science Logic (CSL 2005), 19th Annual Conference of the EACSL*, volume 3634 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2005.

[69] Colin Stirling. Decidability of higher-order matching. *Logical Methods in Computer Science*, 5(3), 2009.

[70] Dejvuth Suwimonteerabuth, Felix Berger, Stefan Schwoon, and Javier Esparza. jmoped: A test environment for java programs. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 164–167. Springer-Verlag, 2007.

[71] Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. jmoped: A java bytecode checker based on moped. In *Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 541–545. Springer-Verlag, 2005.

[72] Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer-Verlag, 1997.

[73] Igor Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 157:234–263, 2001.