Collapsible Pushdown Automata and Recursion Schemes


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 if(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-$n$ $\lambda$-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 Goerd [25, 26] considered the word languages generated by level-$n$ $\lambda$-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 \( \mu \)-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 \(\lambda Y\)-calculus, a formalism equivalent to higher-order recursion schemes. A Krivine machine computes the weak head normal form of a \(\lambda Y\)-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 \(A ::= o \mid A \rightarrow A\). 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 \textit{arity}; the \textit{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 \textit{order} of a type measures the nesting depth on the left of $\to$. We define $\operatorname{ord}(o) = 0$ and $\operatorname{ord}(A_1 \to A_2) = \max(\operatorname{ord}(A_1) + 1, \operatorname{ord}(A_2))$. Thus $\operatorname{ord}(A_1 \to \cdots \to A_n \to o) = 1 + \max\{\operatorname{ord}(A_i) \mid 1 \leq i \leq n\}$. For example, $\operatorname{ord}(o \to o \to o) = 1$ and $\operatorname{ord}((o \to o) \to o \to o) = 3$.

Let $\Sigma$ be a \textit{ranked alphabet} i.e. each $\Sigma$-symbol $f$ has an arity $\operatorname{ar}(f) \geq 0$ which determines its type $o \to \cdots \to o \to o$. Further we assume that each symbol $f \in \Sigma$ is assigned a finite set $\operatorname{Dir}(f) = \{1, \ldots, \operatorname{ar}(f)\}$ of \textit{directions}, and we define $\operatorname{Dir}(\Sigma) = \bigcup_{f \in \Sigma} \operatorname{Dir}(f)$. Let $D$ be a set of directions; a \textit{D-tree} is just a prefix-closed subset of $D^*$, the free monoid of $D$. A $\Sigma$-\textit{labelled ranked and ordered tree} (or simply a \textit{$\Sigma$-labelled tree}) is a function $t : \operatorname{Dom}(t) \to \Sigma$ such that $\operatorname{Dom}(t)$ is a $\operatorname{Dir}(\Sigma)$-tree, and for every node $\alpha \in \operatorname{Dom}(t)$, the $\Sigma$-symbol $t(\alpha)$ has arity $k$ if and only if $\alpha$ has exactly $k$ children and the set of its children is $\{\alpha_1, \ldots, \alpha_k\}$. We write $T^\infty(\Sigma)$ for the set of (finite and infinite) $\Sigma$-labelled trees.

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

### 2.2 Recursion schemes

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

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

- $\Sigma$ is a ranked alphabet of \textit{terminals} (including a distinguished symbol $\bot : o$)
- $\mathcal{N}$ is a finite set of typed \textit{non-terminals}; we use upper-case letters $F, H$, etc. to range over non-terminals
- $S \in \mathcal{N}$ is a distinguished \textit{start symbol} of type $o$
- $\mathcal{R}$ is a finite set of \textit{rewrite rules}, one for each non-terminal $F : (A_1, \ldots, A_n, o)$, of the form
  
  $F \xi_1 \cdots \xi_n \to e$

  where each $\xi_i$ is a variable of type $A_i$, and $e$ is a term in $T(\Sigma \cup \mathcal{N} \cup \{\xi_1, \cdots, \xi_n\})$. Note that the expressions on either side of the arrow are terms of ground type.

The \textit{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 $\Sigma$-labelled trees. Informally the \textit{value tree} \footnote{We refer to the $\Sigma$-labelled tree generated by a recursion scheme as its value tree, because the name is a} $[G]$ of (or the tree \textit{generated by}) a recursion scheme $G$ is a possibly infinite term (of ground type), \textit{constructed from the terminals in $\Sigma$}, that is obtained, starting from the
start symbol $S$, by unfolding the rewrite rules of $G$ \textit{ad infinitum}, replacing formal by actual parameters each time.

To define $\llbracket G \rrbracket$, we first introduce a map $(\cdot)\downarrow : \bigcup A \mathcal{T}_A(\Sigma \cup \mathcal{N}) \rightarrow \bigcup A \mathcal{T}_A(\Sigma)$ that takes a term and replaces each non-terminal, together with its arguments, by $\bot$. We define $(\cdot)\downarrow$ by structural recursion as follows: we let $f$ range over $\Sigma$-symbols, and $F$ over non-terminals in $\mathcal{N}$

$$f\downarrow = f$$
$$F\downarrow = \bot$$

$$(st)\downarrow = \begin{cases} \bot & \text{if } s\downarrow = \bot \\ (s\downarrow t\downarrow) & \text{otherwise.} \end{cases}$$

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

Next we define a one-step reduction relation $\rightarrow_{G}$ which is a binary relation over terms in $\mathcal{T}(\Sigma \cup \mathcal{N})$. Informally, $s \rightarrow_{G} s'$ 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 $\rightarrow_{G}$ is defined by induction over the following rules:

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

The relation $\downarrow_G$ between terms and trees is then defined as follows. Let $s \in \mathcal{T}(\Sigma \cup \mathcal{N})$ and $t \in \mathcal{T}^\infty(\Sigma)$, we define $s \downarrow_G t$ just if

- there is a finite reduction sequence $s = t_0 \rightarrow_{G} \cdots \rightarrow_{G} t_n = t$, and $t$ is a finite tree (none of whose nodes are labelled $\bot$); or
- there is an infinite reduction sequence $s = t_0 \rightarrow_{G} t_1 \rightarrow_{G} t_2 \cdots$ such that $t$ is the defined as the maximal (with respect to its domain) tree such that for every node $\alpha \in \text{Dom}(t)$ there is some $i$ such that for all $j \geq i$, $\alpha \in \text{Dom}(t^+_j)$ and $t(\alpha) = t^+_j(\alpha)$; hence, $t$ may be a finite tree (in which case, some of $t$’s nodes are labelled $\bot$) or an infinite tree.

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

$$\llbracket G \rrbracket = \text{sup}\{ t \in \mathcal{T}^\infty(\Sigma) : S \downarrow_G t \}.$$  

We write $\text{RecTree}_n(\Sigma)$ for the class of value trees $\llbracket G \rrbracket$ where $G$ ranges over order-$n$ recursion schemes.

---

good counterpoint to \textit{computation tree}. We have in mind here the distinction between value and \textit{computation} emphasized by Moggi [49]. The idea is that the value tree is obtained from the computation tree by a (possibly infinite) process of evaluation.


**Example 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:

\[
\begin{align*}
S & \rightarrow H a \\
Hz & \rightarrow F (g z) \\
F \varphi & \rightarrow \varphi (F h)
\end{align*}
\]

The value tree $[G]$ is the $\Sigma$-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 $\varepsilon \cdot 2 \cdot 22 \cdot 221 \cdot 2211 \cdots$.

**Example 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:

\[
\begin{align*}
S & \rightarrow F b c \\
F \varphi \psi & \rightarrow a (\varphi (\psi \sharp)) (F (C_p b \varphi) (C_p c \psi)) \\
C_p \varphi \psi x & \rightarrow \varphi (\psi x)
\end{align*}
\]

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

The value tree $[G]$ is the $\Sigma$-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 \{ $a^\omega$ \} $\cup$ \{ $a^k b^k c^k \sharp^k \mid k \geq 1$ \}. 

7
2.3 The safety constraint

The safety constraint on applicative terms may be regarded as a reformulation of Damm’s derived types \cite{5}. 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 \(\text{ord}(A_1) \geq \text{ord}(A_2) \geq \cdots \geq \text{ord}(A_n)\). 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 \(\text{ord}(s) = \text{ord}(A)\).

**Definition 1.** A rewrite rule \(F x_1 \ldots x_n \rightarrow t\) 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 \(\text{ord}(s) \leq \text{ord}(x_i)\).

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 \cite{47}.

**Example 3.** The scheme \(G_1\) defined in Example \(\ref{example:1}\) is unsafe because of the second rule. The subterm \(gz\) occurs at an operand position and has order 1, but \(z\) has order 0.

Paweł Urzyczyn conjectured that safety is a genuine constraint on expressivity i.e. there is a tree, generated by an order-2 unsafe scheme, which cannot be generated by any safe recursion scheme of any order. This conjecture was recently proved by Paweł Parys \cite{57, 58}.

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\text{-PDA}), in the sense of \cite{40}, in which every non-\(\bot\) 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 \(\ref{section:links}\).

3.1 Stacks with links

Fix a stack alphabet \(\Gamma\) and a distinguished bottom-of-stack symbol \(\bot \in \Gamma\). An \(0\text{-stack}\) (or simply \(0\text{-stack}\)) is just a stack symbol. An \((n + 1)\text{-stack}\) (or simply \((n + 1)\text{-stack}\)) \(s\) is a non-null sequence (written \([s_1 \cdots s_l]\)) of \(n\)-stacks such that every non-\(\bot\) \(\Gamma\)-symbol \(\gamma\) 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 \(\text{\((e + 1)\text{-link}\)\). The order of a stack \(s\) is written \(\text{ord}(s)\).

As usual, the bottom-of-stack symbol \(\bot\) cannot be popped from or pushed onto a stack. Thus we require an \(\text\(0\text{-stack}\) to be a non-null sequence \([\gamma_1 \cdots \gamma_l]\) of elements of \(\Gamma\) such that for all \(1 \leq i \leq l\), \(\gamma_i = \bot\) iff \(i = 1\). We define \(\bot\), the empty \(k\text{-stack}\), as follows: \(\bot_0 = \bot\) and \(\bot_{k+1} = [\bot_k]\).

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

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

$$
\text{pop}_i([s_1 \cdots s_{i+1}]_s) = \begin{cases} 
  [s_1 \cdots s_i] & \text{if } i = \text{ord}(s) \text{ and } l \geq 1 \\
  [s_1 \cdots s_i \text{ pop}_i(s_{i+1})] & \text{if } i < \text{ord}(s)
\end{cases}
$$

By abuse of notation, we set $\text{top}_{\text{ord}(s)+1}(s) = s$. Note that $\text{pop}_i(s)$ is undefined if $\text{top}_{i+1}(s)$ is a one-element $i$-stack. For example $\text{pop}_2([\bot \alpha \beta])$ and $\text{pop}_1([\bot \alpha \beta \bot])$ are both undefined.

There are two kinds of push operations. We start with the order-1 push. Let $\gamma$ be a non-$\bot$ stack symbol and $1 \leq e \leq \text{ord}(s)$, we define a new stack operation $\text{push}_1^{\gamma,e}(s_{i+1})$ that, when applied to $s$, first attaches a link from $\gamma$ to the $(e-1)$-stack immediately below the top $(e-1)$-stack of $s$, then pushes $\gamma$ (with its link) onto the top 1-stack of $s$. Formally for $1 \leq e \leq \text{ord}(s)$ and $\gamma \in (\Gamma \setminus \{\bot\})$, we define

$$
\text{push}_1^{\gamma,e}(s_{i+1}) = \begin{cases} 
  [s_1 \cdots s_i \text{ push}_1^{\gamma,e}(s_{i+1})] & \text{if } e < \text{ord}(s) \\
  [s_1 \cdots s_i s_{i+1} \gamma] & \text{if } e = \text{ord}(s) = 1 \\
  [s_1 \cdots s_i \text{ push}_1^{\gamma}(s_{i+1})] & \text{if } e = \text{ord}(s) \geq 2 \text{ and } l \geq 1
\end{cases}
$$

where

- $\gamma^\bot$ denotes the symbol $\gamma$ with a link to the 0-stack $s_{i+1}$
- $\hat{\gamma}$ denotes the symbol $\gamma$ with a link to the $(e-1)$-stack $s_l$; and we define

$$
\text{push}_1^{\hat{\gamma}}(t_{r+1}) = \begin{cases} 
  [t_1 \cdots t_r \text{ push}_1^{\hat{\gamma}}(t_{r+1})] & \text{if } \text{ord}(t) > 1 \\
  [t_1 \cdots t_{r+1} \hat{\gamma}] & \text{otherwise i.e. } \text{ord}(t) = 1
\end{cases}
$$

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

$$
\text{push}_j([s_1 \cdots s_{i+1}]_s) = \begin{cases} 
  [s_1 \cdots s_{i+1} s_{i+1}] & \text{if } j = \text{ord}(s) \\
  [s_1 \cdots s_i \text{ push}_j(s_{i+1})] & \text{if } j < \text{ord}(s)
\end{cases}
$$

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

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 $\text{top}_1 s$ has a link to $\text{top}_e(s_0)$. Then collapse $s$ is defined to be $s_0$. 


Example 4. When displaying $n$-stacks in examples, we use bent arrows to denote link; however to avoid clutter we shall omit 1-links (indeed by construction they can only point to the symbol directly below), writing e.g. $[[\bot][\bot \alpha \beta]]$ instead of $[[\bot][\alpha \beta \gamma]]$.

Take the 3-stack $s = [[[\bot \alpha]] [[\bot]]]$. We have:

$$\text{push}_{1}^{\beta,2}(s) = [[[\bot \alpha]] [[\bot \alpha \beta]]]$$

$$\text{collapse}(\text{push}_{1}^{\beta,2}(s)) = [[[\bot]]]$$

$$\text{push}_{1}^{\gamma,3}(\text{push}_{1}^{\beta,2}(s)) = [[[\bot \alpha \beta \gamma]]].$$

Then $\text{push}_{2}(\theta)$ and $\text{push}_{3}(\theta)$ are respectively:

$$[[[\bot \alpha]] [[\bot \alpha \beta \gamma]] \text{ and } [[[\bot \alpha \beta \gamma]]].$$

We have $\text{collapse}(\text{push}_{2}(\theta)) = \text{collapse}(\text{push}_{3}(\theta)) = \text{collapse}(\theta) = [[[\bot]]].$

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 $\gamma$ 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 $\gamma \in \Gamma$, $1 \leq e \leq n$ and $k \geq 1$. Purely for convenience, we require that if $\gamma = \bot$ then $e = 1$ and $k = 0$.

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

1. $\text{pop}_{k}$ for each $1 \leq k \leq n$
2. $\text{push}_{j}$ for each $2 \leq j \leq n$
3. $\text{push}_{1}^{\gamma,e}$ for each $1 \leq e \leq n$ and each $\gamma \in (\Gamma \setminus \{\bot\})$, and
4. $\text{collapse}$.

We begin by defining an operation that truncates a stack.

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

We can now define our stack operations. Let $1 \leq e \leq \text{ord}(s)$. We first define $\text{push}_{1}^{\gamma,e}$.
We first define \( \text{push}_1^c \) to aid in the definition of \( \text{push}_1^c(e) \).

\[
\text{push}_1^c([t_1 \cdots t_{r+1}]) = \begin{cases} 
[t_1 \cdots \cdot t_r \text{ push}_1^c(t_{r+1})] & \text{if } \text{ord}(t) > 1 \\
[t_1 \cdots t_{r+1} \gamma] & \text{otherwise i.e. } \text{ord}(t) = 1
\end{cases}
\]

Then we have

\[
\text{push}_1^c(e)(t) = \text{push}_1^{c(e,k)}
\]

assuming \( \text{top}_e(t) = [s_1 \cdots s_{k+1}] \) if \( e > 1 \), and \( \text{top}_2(t) = [s_1 \cdots s_k] \) for \( e = 1 \). We are now ready to define the \( \text{collapse} \) operation by letting

\[
\text{collapse}(s) = \text{bot}_e(s) \quad \text{where } \text{top}_1(s) = \gamma^{(e,k)} \text{ and } k > 0
\]

One can think of the \( \text{collapse} \) operation as a generalisation of the \( \text{pop}_k \) operation for any \( k > 1 \) as we have for any stack \( s \) and any \( k > 1 \) that \( \text{pop}_k(s) = \text{collapse}(\text{push}_1^{c,k}(s)) \) for an arbitrary dummy symbol \( \gamma \).

Now for \( 2 \leq j \leq \text{ord}(s) \):

\[
\text{push}_j([s_1 \cdots s_{l+1}]) = \begin{cases} 
[s_1 \cdots s_{l+1} s_{l+1}] & \text{if } j = \text{ord}(s) \\
[s_1 \cdots s_l \text{ push}_j(s_{l+1})] & \text{if } j < \text{ord}(s)
\end{cases}
\]

. Note that, as an easy consequence of the definitions of the \( \text{push}_1^c(e) \) and the \( \text{push}_k \) operations, a link of order \( e \) always points to a \((e - 1)\)-stack inside the current \( e \)-stack.

**Example 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

\[
\text{push}_1^{\beta,2}(s) = [[[\bot \alpha]]] [[\bot] [\bot \alpha \beta^{(2,1)}]]
\]

\[
\text{push}_1^{\gamma,3}(\text{push}_1^{\beta,2}(s)) = [[[\bot \alpha]]] [[\bot] [\bot \alpha \beta^{(2,1)} \gamma^{(3,1)}]]
\]

\[
\text{push}_2(\text{push}_1^{\gamma,3}(\text{push}_1^{\beta,2}(s))) = [[[\bot \alpha]]] [[\bot] [\bot \alpha \beta^{(2,1)} \gamma^{(3,1)}] [\bot \alpha \beta^{(2,1)} \gamma^{(3,1)}]]
\]

\[
\text{push}_3(\text{push}_1^{\gamma,3}(\text{push}_1^{\beta,2}(s))) = [[[\bot \alpha]]] [[\bot] [\bot \alpha \beta^{(2,1)} \gamma^{(3,1)}]] [[\bot] [\bot \alpha \beta^{(2,1)} \gamma^{(3,1)}]]
\]

and we have

\[
\text{collapse}(\text{push}_2(\text{push}_1^{\gamma,3}(\text{push}_1^{\beta,2}(s)))) = \text{collapse}(\text{push}_3(\text{push}_1^{\gamma,3}(\text{push}_1^{\beta,2}(s)))) = [[[\bot \alpha]]]
\]

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, which are essentially the same as panic automata [11].

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 \( A = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_I \rangle \) where \( A \) is an input alphabet and \( \varepsilon \) is a special symbol, \( \Gamma \) 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 \( \varepsilon \)-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 \( A \) as an \( \varepsilon \)-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 \( \Gamma \); we call \( (q_I, \perp_n) \) the initial configuration.

An \( n \)-CPDA \( A = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_I \rangle \) naturally defines an \( (A \cup \{\varepsilon\}) \)-labelled transition graph \( \text{Graph}(A) := (V, E \subseteq V \times (A \cup \{\varepsilon\}) \times V) \) whose vertices \( V \) are the configurations of \( A \) 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 \( v \xrightarrow{\ell} v' \) to mean that \( (v, a, v') \in E \), and \( v \xrightarrow{\ell \alpha_1, \ldots, \alpha_\ell} v' \) to mean that there exist \( v_0, \ldots, v_\ell \in V \) such that \( v_0 = v, v_\ell = v' \) and \( v_i \xrightarrow{\ell \alpha_1, \ldots, \alpha_\ell} v_{i+1} \) for all \( 0 \leq i < \ell \).

Note that one can transform \( A \), 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 \( \op \) operation\(^3\). In the following we always assume that we are in such a setting.

Example 6. Consider the following 2-CPDA (that actually does not make use of links) \( A = \langle \{1, 2, \varepsilon\}, \{\perp, \alpha\}, \{q_a, q_b, q_c, q_\perp, q_\alpha, q_\beta, q_\gamma\}, \delta, q_\alpha \rangle \) with \( \delta \) as follows (we only give those transitions that may happen):

- \( \delta(q_\alpha, \perp, \varepsilon) = \delta(q_a, \alpha, 2) = (q_a, \push_1^\alpha) \);
- \( \delta(q_a, \alpha, 1) = (q_b, \push_2) \);
- \( \delta(q_b, \alpha, \varepsilon) = \delta(q_b, \alpha, 1) = (q_b, \pop_1) \);
- \( \delta(q_b, \perp, 1) = (q_\alpha, \pop_2) \);
- \( \delta(q_\alpha, \alpha, \varepsilon) = \delta(q_\alpha, \alpha, 1) = (q_\alpha, \pop_1) \);
- \( \delta(q_\alpha, \perp, 1) = (q_\perp, \text{id}) \) where \( \text{id} \) is the operation that leaves the stack unchanged;
- \( \delta(q_\perp, \perp, \perp) \) is undefined.

Then \( \text{Graph}(A) \) is given in Figure [1].

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

\(^3\)This can be done by extending higher-order PDA to allow the annotation of each order-\( k \) stack with the feasibility of the \( \op_a \) operation, which can in turn be transformed into a standard higher-order PDA following the remark on Page 9 of Knapik et al. [35].
A vertex \( v \) in \( \text{Graph}(A) \) is **non-productive** if it is the source of an infinite path labelled by \( \varepsilon^w \) i.e. for every \( k \geq 0 \) there exists \( v_0 \) such that \( v \overset{\varepsilon}{\rightarrow} v_1 \overset{w_1}{\rightarrow} v_2 \overset{\varepsilon}{\rightarrow} v_3 \cdots \) Otherwise \( v \) is said to be **productive**.

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

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

\[
\text{Dom}(t) := \{ w \in A^* \mid \exists v \in V. (q_I, \bot_n) \overset{w}{\rightarrow} v \}.
\]

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

In case \( v_w \) is productive, define \( (q_w, s_w) \) to be the unique configuration with \( v_w \overset{\varepsilon}{\rightarrow} (q_w, s_w) \) that is not the source of an \( \varepsilon \) transition, i.e. that is such that \( (q_w, \text{top}_1(s_w), \varepsilon) \notin \text{dom}(\delta) \).

We can finally define

\[
t(w) := \begin{cases} 
\bot & \text{if } v_w \text{ is non-productive;} \\
\rho(q_w, \text{top}_1(s_w)) & \text{otherwise.}
\end{cases}
\]

Hence there are two kinds of leaves in \( t \): those labelled by symbols in \( \Sigma \) which correspond to dead-ends in \( \text{Graph}(A) \), and those labelled by \( \bot \) which correspond to non-productive vertices in \( \text{Graph}(A) \). 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.
Example 7. Consider the CPDA $A$ from Example 6 and let $\rho(q_a, \cdot) = a$, $\rho(q_b, \cdot) = b$, $\rho(q_c, \cdot) = c$, $\rho(q_{\#}, \cdot) = \#$, where $\cdot$ stands for any stack symbol. Then, the tree generated by $A$ and $\rho$ is the same as the one generated by the order-2 recursion scheme of Example 2.

Remark 1. Thanks to $\varepsilon$-transitions, we can safely assume that the labelling function $\rho$ only depends on the control state, i.e., $\rho: Q \rightarrow \Sigma$ instead of $\rho: Q \times \Gamma \rightarrow \Sigma$, as in Example 7. One can always encode the current top stack symbol in the control state: after each transition, perform an $\varepsilon$-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 $\delta: Q \times \Gamma \times (A \cup \{\varepsilon\}) \rightarrow Q \times \text{Op}_n^\ast$. 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 $\varepsilon$-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 $\varepsilon$-closure (i.e., we contract each $\varepsilon$-labelled edge, merging its source and target vertices). The nodes are labelled according to a function $\rho: \Gamma \rightarrow \Sigma$. 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 $\varepsilon$-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 $\varepsilon$-transitions to taking the $\varepsilon$-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 $\Sigma$ and let $t$ be its value tree. There is an order-$n$ CPDA $A = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_0 \rangle$ and a function $\rho: Q \rightarrow \Sigma$ such that $t$ is the tree generated by $A$ and $\rho$.

(ii) Let $A = \langle A \cup \{\varepsilon\}, \Gamma, Q, \delta, q_0 \rangle$ be an order-$n$ CPDA, and let $t$ be the $\Sigma$-labelled tree generated by $A$ and a function $\rho: Q \rightarrow \Sigma$. There is an order-$n$ recursion scheme over $\Sigma$ 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 5).

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 \( Q = \{q_1, \ldots, q_m\} \) 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 graph 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 \( k \) 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 \( n \). Inductively, for each \( 0 \leq k < n \) we define a type

\[
 k = (k + 1)^m \rightarrow (k + 1)
\]

where, for types \( A \) and \( B \), we write \( A^m \rightarrow B \) as a shorthand for \( A \rightarrow \cdots \rightarrow A \rightarrow B \). In particular, for every \( 0 \leq k < n \), we have

\[
k = (k + 1)^m \rightarrow (k + 2)^m \rightarrow \cdots \rightarrow n^m \rightarrow n
\]

We also introduce a non-terminal \( \text{Void}_k \) of type \( k \) 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 \( [s]_k^p : k \) that represents the behaviour of the topmost \( k \)-stack in \( s \), i.e. \( \text{top}_k+1(s) \). To understand why \( [s]_k^p \) is of type \( k \) 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 \( (k + 1)^m \rightarrow (k + 1) \).

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 \( k \) is defined to be \( (k + 1)^m \rightarrow (k + 1) \).

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

\[
 F_p^{\gamma,e} : e^m \rightarrow 1^m \rightarrow \cdots \rightarrow n^m \rightarrow n
\]

Note that the type of \( F_p^{\gamma,e} \) is non-homogeneous.

For every \( 0 \leq k \leq n \), every state \( p \) and every order-\( n \) stack \( s \) whose topmost stack symbol is \( \text{top}_1(s) = \gamma \) with an \( (e + 1) \)-link, we inductively define the following term of order \( k = (k + 1)^m \rightarrow \cdots \rightarrow n^m \rightarrow n \)

\[
 [s]_k^p = F_p^{\gamma,e}([\text{collapse}(s)]_e^n[\text{pop}_1(s)]_e^n[\text{pop}_2(s)]_e^n \cdots [\text{pop}_k(s)]_e^n)_k
\]
where \([t]^p_k\) is a shorthand for the sequence \([t]^q_1 [t]^q_2 \cdots [t]^q_m\), and if \(\text{pop}_k(s)\) is undefined we set \([\text{pop}_k(s)]^p_k = \text{Void}_1 \cdots \text{Void}_m\).

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 \(k\)-th 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 \(\text{pop}_k^j(s)\) denotes the stack obtained from \(s\) by \(k\) successive applications of the \(\text{pop}_k\) function):

- \(s\) and \(t\) are top\(_1\)-identical just if \(\text{top}_1(s) = \text{top}_1(t)\), and \(\text{collapse}(s)\) and \(\text{collapse}(t)\) are top\(_{e+1}\)-identical where \(\text{top}_1(s)\) has an \((e+1)\)-link
- for \(k > 1\), \(s\) and \(t\) are top\(_k\)-identical just if for every \(j \geq 0\), \(\text{pop}_{k-1}^j(s)\) is defined if and only if \(\text{pop}_{k-1}^j(t)\) is defined, and if so, \(\text{pop}_{k-1}^j(s)\) and \(\text{pop}_{k-1}^j(t)\) 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 \(\text{top}_1(s) = \text{top}_1(t) = (\gamma, e)\) where \(\gamma \in \Gamma\) and \(1 \leq e \leq n\), and that \(\text{collapse}(s)\) and \(\text{collapse}(t)\) are top\(_{e+1}\)-identical. As \(\text{collapse}(s)\) and \(\text{collapse}(t)\) have size bounded by \(N\), by the induction hypothesis, we have \([\text{collapse}(s)]^p_k = [\text{collapse}(t)]^p_k\). Thus it follows immediately that \([s]^p_k = [t]^p_k\). Now take \(k \geq 0\) and assume that the property is established for each \(h \leq k\). 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 \(\text{pop}_h(s)\) (equivalently \(\text{pop}_h(t)\)) is defined then \(\text{pop}_h(s)\) and \(\text{pop}_h(t)\) are top\(_{k+1}\)-identical, and so, by the induction hypothesis, \([\text{pop}_h(s)]^p_k = [\text{pop}_h(t)]^p_k\) 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 \(\text{top}_1(s) = \text{top}_1(t) = (\gamma, e)\) for some \(\gamma \in \Gamma\), and \(\text{collapse}(s)\) and \(\text{collapse}(t)\) are top\(_{e+1}\)-identical. As \(\text{collapse}(s)\) and \(\text{collapse}(t)\) have size bounded by \(N\), by the induction hypothesis, we have \([\text{collapse}(s)]^p_k = [\text{collapse}(t)]^p_k\).

By definition

\[ [s]^{p}_{k+1} = F^{\gamma, e}_p [\text{collapse}(s)]^p_k [\text{pop}_1(s)]^p_k \cdots [\text{pop}_{k+1}(s)]^p_k \]

and

\[ [t]^{p}_{k+1} = F^{\gamma, e}_p [\text{collapse}(t)]^p_k [\text{pop}_1(t)]^p_k \cdots [\text{pop}_{k+1}(t)]^p_k \]

Now for any \(n\)-stack \(r\) define \(j_r\), be the maximal \(j\) such that \(\text{pop}_{n+1}^j(r)\) is defined. In particular we have \(j_s = j_t\). If \(j_s = 0\), \([\text{pop}_{k+1}(s)]^p_k = [\text{pop}_{k+1}(t)]^p_k = \text{Void}_{k+1} \cdots \text{Void}_{k+1}\), and so \([s]^{p}_{k+1} = [t]^{p}_{k+1}\). If \(j_s > 0\), note that \(j_{\text{pop}_{k+1}(s)} = j_{\text{pop}_{k+1}(t)} = j_s - 1\) and \(\text{pop}_{k+1}(s)\) and \(\text{pop}_{k+1}(t)\) are top\(_{k+1}\)-identical. Thus, inductively (on \(j_s\)), we have \([\text{pop}_{k+1}(s)]^p_k = [\text{pop}_{k+1}(t)]^p_k\) 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

\[
F_{\gamma, e}^p \Phi \Psi_1 \cdots \Psi_n \xrightarrow{\theta} \Xi_\theta
\]

where \( 0 \leq j \leq n \) and we have \( \Psi_j = \Psi_{j,1} \cdots \Psi_{j,m} \) is a sequence of variables, with each \( \Phi_i : e \). The shape of \( \Xi_\theta \) depends on \( op \), as shown in Table 1, where \( (F_{\gamma, e}^p \Phi \Psi_1 \cdots \Psi_k | i) \) is a shorthand for the sequence

\[
F_{\gamma_1} \Phi \Psi_1 \cdots \Psi_k \quad \cdots \quad F_{\gamma_m} \Phi \Psi_1 \cdots \Psi_k : k^m
\]

The preceding labelled rewrite rules induce a \( \theta \)-indexed family of outermost labelled one-step transition relations \( \xrightarrow{\theta} \subseteq T^0(\mathcal{N}_A) \times T^0(\mathcal{N}_A) \), where \( \theta \) ranges over \( Q \times Op_n \). Informally, \( M \xrightarrow{\theta} M' \) 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 \( F_{\gamma, e}^p \Phi \Psi_1 \cdots \Psi_k | i \), we have the rule scheme

\[
F_{\gamma, e}^p \Phi M_1 \cdots M_n \xrightarrow{\theta} \Xi_\theta[\Phi/\Psi_1, \cdots, M_n/\Psi_n]
\]

where \( L, M_1, \cdots, M_n \) range over sequences of terms that respect the type of \( F_{\gamma, e}^p \).

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 \( \theta \)-compatible just if \( \theta \) 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 \( \mathcal{A} \).

<table>
<thead>
<tr>
<th>Cases of ( op )</th>
<th>Corresponding ( \Xi_\theta ) where ( \theta = (q, op) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{push}_1^{\gamma, e'} )</td>
<td>( F_{\gamma}^{\gamma, e'} \Phi \Psi_1(i) \Psi_2 \cdots \Psi_n )</td>
</tr>
<tr>
<td>( \text{push}_k )</td>
<td>( F_{\gamma}^{\gamma, e} \Phi \Psi_1 \cdots \Psi_{k-1} \Psi_k(i) \Psi_{k+1} \cdots \Psi_n )</td>
</tr>
<tr>
<td>( \text{pop}_k )</td>
<td>( \Phi_k \Psi_{k-1} \cdots \Psi_n )</td>
</tr>
<tr>
<td>( \text{collapse} )</td>
<td>( \Phi \Psi_{e-1} \cdots \Psi_n )</td>
</tr>
</tbody>
</table>
Proposition 1. Let \((p, s)\) be a configuration of \(A\) and \(\theta = (q, op) \in Q \times Op_n\). If \((p, s)\) is \(\theta\)-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 \(\theta\)-compatible. Set \(C^q = [\text{collapse}(s)]_e^q : e\), and \(T_k^n = [\text{pop}_k(s)]_k^n : k\) for every \(1 \leq i \leq m\) and every \(1 \leq k \leq n\). Then

\[
[s]_n^p = F_p^{\gamma,e} C^q_1 \cdots C^q_m \ T_1^q \cdots T_m^q \ T_1^{q_1} \cdots T_n^{q_n}
\]

- Assume \(op = \text{push}_{i}^{\gamma,e}\). By definition we have

\[
[\text{push}_{i}^{\gamma,e}(s)]_n^q = F_q^{\gamma,e} [\text{collapse}(\text{push}_{i}^{\gamma,e}(s))]_e^q \text{ and } [\text{pop}_1(\text{push}_{i}^{\gamma,e}(s))]_n^q = T_1^q.\]

We have \(\text{collapse}(\text{push}_{i}^{\gamma,e}(s)) = \text{pop}_1(s)\), hence \([\text{collapse}(\text{push}_{i}^{\gamma,e}(s))]_e^q = T_1^q\). Further we have \([\text{pop}_1(\text{push}_{i}^{\gamma,e}(s))]_n^q = F_q^{\gamma,e} C^n T_1^n\). Finally, for each \(j > 1\), we have \([\text{pop}_j(\text{push}_{i}^{\gamma,e}(s))]_n^q = T_j^n\).

Therefore, we have

\[
[\text{push}_{i}^{\gamma,e}(s)]_n^q = F_q^{\gamma,e} T_1^q (F_q^{\gamma,e} C^n T_1^n) \cdots (F_q^{\gamma,e} C^n T_1^n) T_2^n \cdots T_n^n
\]

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 \([s]_n^p \xrightarrow{\theta} t\). Hence one has \([s]_n^p \xrightarrow{\theta} [\text{push}_{i}^{\gamma,e}(s)]_n^q\).

- Assume \(op = \text{push}_k\). By definition we have

\[
[\text{push}_k(s)]_n^q = F_q^{\gamma,e} [\text{collapse}(\text{push}_k(s))]_e^q [\text{pop}_1(\text{push}_k(s))]_n^q \cdots [\text{pop}_n(\text{push}_k(s))]_n^q
\]

Note that we used the fact that the top element in \(\text{push}_k(s)\) is the same as that in \(s\) i.e. it is \(\gamma\) and has an \((e + 1)\)-link. Now if \(e \leq k\), \(\text{collapse}(\text{push}_k(s))\) and \(\text{collapse}(s)\) are \(\text{top}_{e+1}\)-identical; hence, thanks to Lemma [1]

\([\text{collapse}(\text{push}_k(s))]_n^q = [\text{collapse}(s)]_n^q = C^n\).

If \(e > k\), \(\text{collapse}(s) = \text{collapse}(\text{push}_k(s))\); hence we also have

\([\text{collapse}(\text{push}_k(s))]_n^q = C^n\).

Next, for \(j < k\), \([\text{pop}_j(\text{push}_k(s))]_j^q = \text{top}_{j+1}\)-identical; hence, thanks to Lemma [1]
\([\text{pop}_j(\text{push}_k(s))]_j^q = [\text{pop}_j(s)]_j^q = T_j^n\). Further we have \([\text{pop}_k(\text{push}_k(s))]_n^q = F_q^{\gamma,e} C^n T_1^n \cdots T_k^n\) for every \(1 \leq i \leq m\). Finally, for every \(j > k\), we have \([\text{pop}_j(\text{push}_k(s))]_n^q = \text{pop}_j(s)\); hence \([\text{pop}_j(\text{push}_k(s))]_n^q = T_j^n\). Therefore we have

\[
[\text{push}_k(s)]_n^q = F_q^{\gamma,e} C^n T_1^n \cdots T_k^n
\]

\[
(F_q^{\gamma,e} C^n T_1^n \cdots T_k^n) (F_q^{\gamma,e} C^n T_1^n \cdots T_k^n) \cdots (F_q^{\gamma,e} C^n T_1^n \cdots T_k^n) T_{k+1}^n \cdots T_n^n
\]
On the other hand, it follows syntactically from the definition of $\theta$ that the right-hand side of the preceding equation is the term $t$ such that $[s]_n^p \xrightarrow{\theta} t$. Hence one has $[s]_n^p \xrightarrow{\theta} [\text{push}_k(s)]_n^q$.

- Assume $op = \text{pop}_k$. By definition we have

$$[\text{pop}_k(s)]_n^q = \mathcal{F}_q^n [\text{collapse}(\text{pop}_k(s))]_r^q [\text{pop}_1(\text{pop}_k(s))]_1^q \cdots [\text{pop}_n(\text{pop}_k(s))]_n^q$$

where the top$_1$ element in $\text{pop}_k(s)$ is a $\gamma'$ and has an $(e' + 1)$-link. It follows that

$$[\text{pop}_k(s)]_n^q = [\text{pop}_k(s)]_n^q [\text{pop}_{k+1}(\text{pop}_k(s))]_n^q \cdots [\text{pop}_n(\text{pop}_k(s))]_n^q$$

For every $j > k$, we have $\text{pop}_j(\text{pop}_k(s)) = \text{pop}_j(s)$, hence $[\text{pop}_j(\text{pop}_k(s))]_n^q = T_j^n$. Therefore we have $[\text{pop}_k(s)]_n^q = T_k^n T_{k+1}^n \cdots T_n^n$. On the other hand, it follows syntactically from the definition of $\theta$ that the right-hand side of the preceding equation is the term $t$ such that $[s]_n^p \xrightarrow{\theta} t$. Hence one has $[s]_n^p \xrightarrow{\theta} [\text{pop}_k(s)]_n^q$.

- Assume $op = \text{collapse}$. By definition we have

$$[\text{collapse}(s)]_n^q = \mathcal{F}_q^n [\text{collapse}(\text{collapse}(s))]_r^q [\text{pop}_1(\text{collapse}(s))]_1^q \cdots [\text{pop}_n(\text{collapse}(s))]_n^q$$

where the top$_1$ element in $\text{collapse}(s)$ is a $\gamma'$ and has an $(e' + 1)$-link. Equivalently, one has

$$[\text{collapse}(s)]_n^q = [\text{collapse}(s)]_n^q [\text{pop}_{e+1}(\text{collapse}(s))]_n^q \cdots [\text{pop}_n(\text{collapse}(s))]_n^q$$

For every $j > e$ we have $\text{pop}_j(\text{collapse}(s)) = \text{pop}_j(s)$, hence $[\text{pop}_j(\text{collapse}(s))]_n^q = T_j^n$. Therefore we have $[\text{collapse}(s)]_n^q = C^n T_{e+1}^n \cdots T_n^n$. On the other hand, it follows syntactically from the definition of $\theta$ that the right-hand side of the preceding equation is the term $t$ such that $[s]_n^p \xrightarrow{\theta} t$. Hence one has $[s]_n^p \xrightarrow{\theta} [\text{collapse}(s)]_n^q$.

\[ \square \]

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

$$\mathcal{N} = \{ \mathcal{F}_{p}^{e+} \mid \gamma \in \Gamma, \ 1 \leq e \leq n, \ p \in Q \} \cup \{ \text{Void}_i \mid 1 \leq i \leq n \},$$

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

### 5.4 The recursion scheme $G_A$ determined by a CPDA $A$

Fix some integer $d \geq 1$ and let $[d]$ denote $\{1, \ldots, d\}$. Fix an $n$-CPDA $A = ([d] \cup \{\varepsilon\}, \Gamma, Q, \delta, q_1)$ and a function $\rho : Q \times \Gamma \rightarrow \Sigma$, and let $Q = \{q_1, \ldots, q_m\}$. Let $t$ be the tree generated by $A$ and $\rho$ as defined in Section 5.3

We define from $A$ and $\rho$ 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 $A$. 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 $\text{Graph}(A)$. 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 $A$ and $\rho$ is defined to be $G_{A,\rho} = \langle \Sigma, N, R, S \rangle$ (written $G_A$ if $\rho$ is clear) where

$$N = \{ F_{\gamma,e}^{i} \mid \gamma \in \Gamma, \ 1 \leq e \leq n, \ p \in Q \} \cup \{ \text{Void}_i \mid 1 \leq i \leq n \}$$

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

$$S \rightarrow \rho(q_1, \perp) \quad \text{if } ar(\rho(q_1, \perp)) = 0$$

$$S \rightarrow F_{q_1}^{1,1} \text{Void}_1 \cdots \text{Void}_n \quad \text{otherwise}$$

and

$$F_{\gamma,e}^{i} F_{\psi_1} \cdots F_{\psi_n} \rightarrow \Xi$$

if $\delta(p,\gamma,\varepsilon) = \theta$ is defined

$$F_{\gamma,e}^{i} F_{\psi_1} \cdots F_{\psi_n} \rightarrow \rho(p,\gamma) \Xi_{\theta_1} \cdots \Xi_{\theta_r}$$

otherwise, where $r = ar(\rho(p,\gamma))$ 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 $\gamma$ where $\{ a \in [d] \cup \{ \varepsilon \} \mid (p,\gamma,a) \in \text{Dom}(\delta) \} = \emptyset$. Indeed, one still needs to produce a terminal for them as they correspond to productive leaves in the tree obtained from $\text{Graph}(A)$ 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 $A$ be a tree-generating CPDA, and $G_A$ be the recursion scheme determined by $A$. Then the CPDA and the recursion scheme generate the same $\Sigma$-labelled tree.

**Proof.** The proof follows from Proposition 1 the definition of $G_A$ 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 $S \rightarrow_{G_A}^* t$ where $\rightarrow_{G_A}^*$ denotes the transitive closure of $\rightarrow_{G_A}$.

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 $i \in \{ 1, \ldots, \ell \}$ such that $t = ft_1 \cdots t_\ell$ 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 $\text{location}_t(s)$ (or simply $\text{location}(s)$ if $t$ is clear), which is a sequence defined by

$$\text{location}_t(s) := \begin{cases} \varepsilon & \text{if } s = t \\ f, i, \text{location}_{t_i}(s) & \text{otherwise, where } t = ft_1 \cdots t_\ell \text{ and } s \text{ is a subterm of } t_i \end{cases}$$

One can easily characterise those terms that can be derived from $S$ in $\rightarrow_{G_A}^*$. Indeed we have $S \rightarrow_{G_A}^* t$ if and only if either $t = S$ or for every subterm $t'$ of $t$ such that $\text{location}_t(t') =$
first introduce several concepts we need for the rest of the section. We write \([n]\) as a shorthand for \(\{1, \ldots, n\}\) and \([n]_0\) for \(\{0, \ldots, n\}\). Fix a ranked alphabet \(\Sigma\). Typically\(^4\) \(\text{Dir}(f) = [\text{ar}(f)]\) and we always have \(|\text{Dir}(f)| = \text{ar}(f)\) for each \(\Sigma\)-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 \textit{long transform}. For each \(G\)-rule:

\(^{4}\)The only exception is the symbol \(@_A\) of the auxiliary alphabet \(\Lambda_G\), where we have \(\text{Dir}(@_A) = [\text{ar}(@_A) - 1]_0\).
1. **Expand the right-hand side to its \( \eta \)-long form.** *I.e.* we hereditarily \( \eta \)-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 \( \uparrow s_1 \cdots s_m \) where \( \uparrow \) is either a variable (i.e. some \( \xi_j \)) or a non-terminal or a terminal. Suppose \( \uparrow s_1 \cdots s_m : (A_1, \cdots, A_n, o) \). We define

\[
\uparrow s_1 \cdots s_m \downarrow = \lambda \overline{\varphi} : \uparrow s_1 \downarrow \cdots \uparrow s_m \downarrow \gamma \varphi_1 \gamma \cdots \gamma \varphi_n \gamma
\]

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 \( \uparrow s_1 \downarrow \cdots \uparrow s_m \downarrow \).

For example the \( \eta \)-long form of \( g_\cdot o \) is \( \lambda.g(\lambda.a) \); we shall see that the “dummy lambda-abstraction”\(^5\) \( \lambda.a \) (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 \( De_1 \cdots e_n \), where \( D : (A_1, \cdots, A_n, o) \) is a non-terminal and \( n \geq 1 \) (i.e. \( D \) has order at least 1), by \( @_A De_1 \cdots e_n \) where \( A = ((A_1, \cdots, A_n, o), A_1, \cdots, A_n, o) \) 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 \( D, e_1, \cdots, e_n \).

3. **Curry the rewrite rule.** *I.e.* we transform the rule \( F\varphi_1 \cdots \varphi_n \to \lambda.e' \) to

\[
F \to \lambda\varphi_1 \cdots \varphi_n.e'.
\]

In case \( n = 0 \), note that the curried rule has the form \( F \to \lambda.e' \).

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

**Example 8.** We revisit the recursion scheme of Example[7] and illustrate the long transform:

\[
\begin{align*}
G : & \quad \begin{cases}
S \to Ha \\
Hz \to F(gz) \\
F\varphi \to \varphi(F.h)
\end{cases} & \quad \implies \quad \overline{G} : & \quad \begin{cases}
S \to \lambda@ H(\lambda.a) \\
Hz \to F(\lambda.y.g(\lambda.z)(\lambda.y)) \\
F\varphi \to \varphi(\lambda.(F.h))
\end{cases}
\end{align*}
\]

For instance, the right hand side of the third rule is \( \lambda.\varphi(\lambda.(\lambda.F(\lambda.(x.h)(\lambda.x)))) \) after the first step, and \( \lambda.\varphi(\lambda.(\lambda.\varphi(\lambda.(\lambda.F(\lambda.(x.h)(\lambda.x))))) ) \) 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 \( \Lambda_G \), which is \( \Sigma \) augmented by certain variables and lambdas (of the form \( \lambda\overline{\xi} \) which is a short hand for \( \lambda\xi_1 \cdots \xi_n \) where \( n \geq 0 \) but regarded as terminals. The alphabet \( \Lambda_G \) is a finite subset of the set

\[
\Sigma \cup \text{Var} \cup \{ @_A | A \in \text{ATypes} \} \cup \{ \lambda\overline{\xi} | \overline{\xi} \subseteq \text{Var} \}
\]

\(^5\)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].
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 $\Lambda_G$ as follows:

- variable symbol $\varphi : (A_1, \cdots, A_n, o)$ in $\text{Var}$ has arity $n$
- long-apply symbol $\@_A$ where $A = ((A_1, \cdots, A_n, o), A_1, \cdots, A_n, o)$ has arity $n + 1$
- lambda symbol $\lambda \xi$ has arity 1, for every list of variables $\xi \subseteq \text{Var}$.

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

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

For technical reasons (to be clarified shortly), the leftmost child of an $\@$-labelled node $\alpha$ is in direction 0 (i.e. it is $\alpha$’s 0-child); for all other nodes, the leftmost child is in direction 1. The non-terminals of $\mathcal{G}$ 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 $[\mathcal{G}]$ of the order-0 recursion scheme $\mathcal{G}$. It follows that $\lambda(G)$ is a regular tree.$^6$

A $\Lambda$-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 $l : V \rightarrow \Lambda$ with $\Lambda$ 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 $i = 1, \ldots, \text{ar}(l(v))$.

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

The unfolding of $\mathcal{K}$ is the $\Lambda$-labelled ranked tree $t : \text{Dom}(t) \rightarrow \Lambda$ such that $\text{Dom}(t)$ 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 $\varepsilon$ 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 $\mathcal{G}$. We define the HORS graph $Gr(G)$ to be the $\Lambda_G$-labelled DDG determined by $G$

$$Gr(G) = \langle V, E \subseteq V \times V, \lambda_G : V \rightarrow \Lambda_G, \ v_0 \in V \rangle$$

that is obtained by the following procedure:

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

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

$$D_F = \langle V_F, E_F \subseteq V_F \times V_F, l_F : V_F \rightarrow \Lambda_{G,\mathcal{G}}, \ rt_F \rangle$$

$^6$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.
given by the $\Lambda_G$-labelled tree that is determined by the right-hand side of the rule, namely, $\lambda \varphi_1 \cdots \varphi_n.e$. In particular, $rt_F$ is the root of that tree and we have $l_F(rt_F) = \lambda \varphi_1 \cdots \varphi_n$ with reference to the rule $F$ given above.

3. First for each $F$ in $\mathcal{N}_G$ we define

$$E_F = \bigcup_{H \in \mathcal{N}_G} l_H^{-1}(\{ F \})$$

and

$$\mathcal{E} = \bigcup_{F \in \mathcal{N}_G} \mathcal{E}_F$$

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

Formally we have the following.

- The vertices $V$ of $Gr(G)$ are defined by

$$V = \bigcup_{F \in \mathcal{N}_G} V_F \setminus \bigcup_{F \in \mathcal{N}_G} E_F$$

- The root $v_0$ of $Gr(G)$ is $rt_S$, where $S$ is the start symbol of $G$.

- The vertex-labels are defined by

$$\lambda_G(v) = \begin{cases} 
l_F(rt_F) & \text{if } v = E_F \text{ for some } F \in \mathcal{N}_G \\
l_H(v) & \text{otherwise, where } v \text{ is a vertex in } V_H \end{cases}$$

- The edges $E$ of $Gr(G)$ are defined by

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

and the edge-labels of $Gr(G)$ are inherited from the edge-labels of the component DDGs $D_F$ according to how vertices and edges were merged.

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

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

Fix a HORS graph $Gr(G) = \langle V, E, \lambda_G, v_0 \rangle$. We shall call a vertex of $Gr(G)$ **prime** just if it is the 0-child 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

---

7 Note that if $Gr(G)$ 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.
Collapsible Pushdown Automata and Recursion Schemes

M. Hague, A. S. Murawski, C.-H. L. Ong, O. Serre

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

(so that the root has depth 0). Let $u$ be a vertex. We define $\text{pred}(u) = \{ u' \in V : (u',u) \in E \}$ i.e. the set of predecessors of $u$. For every vertex $u$ labelled by a variable $\varphi_i$ (say), its binder, written $\text{binder}(u)$, is the vertex that is labelled $\lambda \varphi$, where $\varphi$ is a list of variables that contains $\varphi_i$. (Since bound variables are renamed to prevent any clash in the construction of $G$, every variable vertex in $\text{Gr}(G)$ has a unique binder.) We say that $u$ is the $i$-parameter of $\text{binder}(u)$ just if $\varphi_i$ is the $i$th-item of the list $\varphi$. The span of the variable vertex $u$ is defined to be the depth of $u$ minus the depth of $\text{binder}(u)$.

We note the following features of HORS graphs:

(i) Except the root and possibly some prime vertices, every vertex $u$ has a unique predecessor $v$. 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 predecessor 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 $\text{binder}(u)$ 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 $\Lambda_G$-label $\lambda_G(u)$. 

25
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 \( \xi \). 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 \( \lambda \xi \) for some list \( \xi \) that contains \( \xi \).

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 “\( n' \) \( i \)-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 \( \xi_i \), say) is \( i \)-enabled by its binder (labelled by some \( \xi_i \), say) where \( \xi_i \) is the \( i \)-th element of the list \( \xi \).

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 \( \lambda \)), and all nodes labelled by a long-apply or a \( \Sigma \)-symbol.

Enabling permits us to define the notion of *justified sequence* over the computation tree. A unjustified 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-occurrence \( n_0 \) in the sequence such that \( n_0 \vdash_i n \) for some \( i \). We say that the node-occurrence \( n \) is justified by the node-occurrence \( n_0 \) in the sequence. We use the notation

\[
\cdots n_0 \cdots n \cdots
\]

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.  

Let \( t \) be a justified sequence and let \( n \) be some occurrence of a node in \( t \). Then we write \( t_{\leq n} \) (resp. \( t_{< n} \)) to mean the prefix of \( t \) 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).  

---

\(^9\) Justified sequences were first introduced to represent plays in dialogue games between two players, O and P [39]. 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 $\varepsilon$, is a traversal.

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

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

(Var) If $t_{n_{1}}\cdots n_{i}\xrightarrow{n_{i}'}$ is a traversal for some sequence $t$ and some lambda node $n'$ (labelled by some $\lambda\xi$) and some variable node $n''$ (labelled by $\xi_{i}$, the $i$-th variable in $\xi$), so is $tn\xrightarrow{n''}$ for each node $n''$. Note that $n''$ is always a lambda node.

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

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 deterministic digraph

$$Tr(G) = \langle V, E \subseteq V \times (\text{Dir}(\Sigma) \cup \{\varepsilon\}) \times V, \lambda : V \rightarrow (\Sigma \cup \{\sharp\}), v_0 \rangle$$

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 $\varepsilon$.

Note that $Tr(G)$ is a deterministic tree. By taking its $\varepsilon$-closure as explained below, we obtain a $\Sigma$-labelled ranked tree. More precisely, the traversal tree of $G$, denoted $TrTree(G)$, is obtained as follows from $Tr(G)$. For every $i \in \text{Dir}(\Sigma)$, add an $i$-labelled edge from $v_1$ to $v_2$

---

10In fact we have a small abuse of terminology and notation here as in our original definition of a DDG we do not allow $\varepsilon$-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 $V \times (\text{Dir}(\Sigma) \cup \{\varepsilon\}) \times V$. 

27
Theorem 4 (Correspondence Theorem). For every recursion scheme $G$, we have $[G] = \text{TrTree}(G)$.

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

- $^\tau \lambda \gamma = \lambda$ for a dummy lambda $\lambda$.

- $^\tau t n' \cdots n \gamma = ^\tau t' n \cdots n$ 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.

- $^\tau t n \gamma = ^\tau t' n$ whenever $n$ is a non-lambda node.

In the second clause above, if in $t n' \cdots n$ the non-lambda node $n'$ has a pointer to some node-occurrence $l$ (say) in $t$, and if $l$ appears in $^\tau t \gamma$, then in $^\tau t' n \cdots n$ 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-occurrence $l$ (say) in $t$ and if $l$ appears in $^\tau t \gamma$, then in $^\tau t' n$ 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[55] Let $t$ be a finite traversal over a computation tree $\lambda(G)$. Then the following hold.

- $^\tau t \gamma$ is a well-defined justified sequence.

- $^\tau t \gamma$ 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 $\text{Gr}(G)$ we can associate with every sequence of nodes in $\lambda(G)$ a unique sequence of vertices in $\text{Gr}(G)$. 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 $\text{Gr}(G)$ (in this case, it is easy to reconstruct the corresponding traversal over $\lambda(G)$). Note that traversals over $\text{Gr}(G)$ could equivalently be defined by stating the rules in Table 2 in the framework of $\text{Gr}(G)$.

6.2 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, CPDA($G$), and will show that its transition graph Graph(CPDA($G$)) is trace-equivalent with $\text{Tr}(G)$. As a byproduct, unfolding the $\varepsilon$-closure of Graph(CPDA($G$)) leads to the same tree as $\text{TrTree}(G) = \lambda(G)$, equivalently CPDA($G$) generates the same tree as $G$.

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

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

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 CPDA($G$), we shall write $\text{push}_{a,1}$ as $\text{push}_{a}^1$, 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 CPDA($G$).

**Definition 4.** The transform 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 $[\cdots [\downarrow v_0] \cdots ]$ i.e. $\text{push}_{1}^{v_0} \downarrow_n$, where $v_0$ is the root of $\text{Gr}(G)$. Let $u$ range over the stack symbols of CPDA($G$). For ease of explanation, we define the transition map $\delta$ 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 $\Lambda_G$) 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 Graph(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 $K_1 = \langle V_1, E_1 \subseteq V_1 \times \Pi \times V_1, l : V_1 \rightarrow \Lambda, r_1 \rangle$ and $K_2 = \langle V_2, E_2 \subseteq V_2 \times \Pi \times V_2, l : V_2 \rightarrow \Lambda, r_1 \rangle$ be two graphs with the same alphabet for labelling vertices (resp. edges). We say that $K_1$ and $K_2$ are trace-equivalent just if for every $x \in \{1, 2\}$, for every path in $K_x$ that starts from the root $r_x$, there is a (unique) corresponding path in $K_\pi$ (here $\pi = 2$ if $x = 1$ and $\pi = 1$ otherwise) with the same label in $\Pi^*$. Moreover the terminal vertices of both paths are labelled by the same element in $\Lambda$.

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

The following proposition follows by definition.

29
Table 3: Definition of the transform $\text{CPDA}(G)$.

If $u$’s label is not a variable, the action is just a $\text{push}_1^v$, where $v$ is an appropriate child of the node $u$. Precisely:

- **(A)** If the label is an @ then $\delta(u, \varepsilon) = \text{push}_{E_0}^1(u)$.

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

- **(L)** If the label is a lambda then $\delta(u, \varepsilon) = \text{push}_{E_1}^1(u)$.

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

- **(V₁)** If the variable has order $l \geq 1$, then

  $$\delta(u, \varepsilon) = \begin{cases} 
  \text{push}_{n-l+1}^p; \text{pop}_{\varepsilon}^{p+1}; \text{push}_{E_i}^1(\text{top}_1), n-l+1 & \text{if } j = 0 \\
  \text{push}_{n-l+1}^p; \text{pop}_{\varepsilon}^{p}; \text{collapse}; \text{push}_{E_i}^1(\text{top}_1), n-l+1 & \text{otherwise}
  \end{cases}$$

  where $\text{pop}_{\varepsilon}^p$ means the operation $\text{pop}_{\varepsilon}$ iterated $p$ times, and $\text{push}_{E_i}^1(\text{top}_1), k$ is defined to be the operation $s \mapsto \text{push}_{E_i}^1(\text{top}_1, s), k$.

- **(V₀)** Otherwise (i.e. the variable has order 0)

  $$\delta(u, \varepsilon) = \begin{cases} 
  \text{pop}_{\varepsilon}^{p+1}; \text{push}_{E_i}^1(\text{top}_1) & \text{if } j = 0 \\
  \text{pop}_{\varepsilon}^{p}; \text{collapse}; \text{push}_{E_i}^1(\text{top}_1) & \text{otherwise}.
  \end{cases}$$

**Proposition 3.** Let $K_1$ and $K_2$ be two trace-equivalent labelled rooted deterministic digraphs (possibly with $\varepsilon$-labelled edges). Then the trees obtained by unfolding the $\varepsilon$-closure of those two digraphs are the same.

Let $\text{Graph}_\varepsilon(\text{CPDA}(G))$ be the vertex labelled version of $\text{Graph}(\text{CPDA}(G))$ (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 $\Sigma$ and by $\sharp$ otherwise. We have the following key result (to be proved later).

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

An important consequence of Theorem 5 is the following.

**Theorem 6** (Equi-Expressivity 2). For every order-$n$ recursion scheme $G$, $\text{CPDA}(G)$ (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, $\text{Graph}_\varepsilon(\text{CPDA}(G))$ and $\text{Tr}(G)$ are trace-equivalent. By definition, the tree generated by $\text{CPDA}(G)$ and the identity function,
collapsible pushdown automata and recursion schemes M. Hague, A. S. Murawski, C.-H. L. Ong, O. Serre

coincides with that obtained by unfolding the ε-closure of Graphℓ(\text{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 Graphℓ(\text{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 Graphℓ(\text{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 \(Σ \cup \{♯\}\). 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 ♯) or
- by \(ar(f)\) edges with labels in \(\{1, \cdots, ar(f)\}\) (if the final vertices of the paths are labelled by \(f \in Σ\)).

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 \(1 ≤ i ≤ n\), the sequence of node labels corresponding to the top 1-stack, written \(λ_G(top_2(s_i))\), is the P-view of \(t_i\), i.e.

\[ λ_G(top_2(s_i)) = \lceil t_i \rceil. \]

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

\[ \$ \xrightarrow{w} \lambda \]

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

\[ \lambda \bar{v} \cdots x \]

where \(x\) is an order-0 variable symbol and \(\bar{v}\) is a list of variables in which \(x\) occurs. Finally, we remove all pointers from \(\hat{t}_i\). See Example 10 for an illustration of this construction. We then transform each \(n\)-stack \(s_i\) to a sequence of nodes \(s_{i_j}\), which will be shown to coincide with \(\hat{t}_i\).
Remark 8. Note that 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 \( s_i \) 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)

   \[
   [\tiny{[[\cdots] \cdots][\cdots] \cdots[[\cdots][\cdots]]}]\]

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 \( s_i \) – 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 \( s \) 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 \( s \) of stack symbols is defined as follows.

\[
\begin{align*}
\mathcal{S} &= \begin{cases} 
\varepsilon & \text{top}_2(s) = [ ], \\
(\text{pop}_1(s)) \lambda_G(u) & \text{top}_1(s) = u \text{ and } u \text{ has a 1-link,} \\
(\text{collapse}(s)) \lambda_G(u) & \text{top}_1(s) = u \text{ and } u \text{ has a } k \text{-link with } k > 1.
\end{cases}
\end{align*}
\]

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

**Definition 6.** Let \( G \) be an order-\( n \) recursion scheme, let \( s \) be a reachable configuration of 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) \( \lambda_G(\text{top}_2(s)) = \text{"} t \text{"} \).

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

(c) Suppose \( \text{top}_2(s) = [v_1, \cdots, v_n] \). Let \( v_i', \cdots, v_n' \) be the respective occurrences of \( v_1, \cdots, v_n \) in \( t \) that contribute to \( \text{"} t \text{"} \). Then \( \text{pop}_1^{n-i}(s) \) computes \( t_{\leq v_i'} \) for every \( 1 \leq i < n \).
(d) Using the same notation as in (c), suppose \( \psi \) has a link to an \( l \)-stack \( \sigma \). Let \( s_\sigma \) be the prefix of \( s \) such that \( \sigma \) is its top \( l \)-stack, i.e. \( s_\sigma = \text{collapse}(\text{pop}_{l-1}(s)) \). Then \( s_\sigma \) computes \( t_{<\psi} \).

Note that the definition is not circular, since \( t_{<\psi} \) (1 ≤ \( i \) < \( n \)) and \( t_{<\psi'} \) (1 ≤ \( i \) ≤ \( n \)) are strictly shorter than \( t \). In what follows we shall blur the distinction between \( \psi \) and its occurrence \( \psi' \), 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, \( s' = \text{push}_k(s) \) and \( k \geq 2 \). Given an \( n \)-stack \( s'' \) such that \( s < s'' \leq s' \), let \( s_{\psi''} \) 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 \( s_{\psi''} \) computes \( t \) for any traversal \( t \).

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

For the inductive step, we need to show that \( s'' \) and \( s_{\psi''} \) compute the same traversals. To that end, observe that \( \text{top}_{k}(s'') = \text{top}_{k}(s_{\psi''}) \), because – according to the definition of \( \text{push}_k \) – \( \text{top}_{k}(s'') \) is a clone of \( \text{top}_{k}(s_{\psi''}) \). Note also that \( \text{push}_k \) preserves links. Hence, \( s'' = s_{\psi''} \).

Thus, conditions (a) and (b) are fulfilled for the same traversal. For (c) and (d), recall that \( \text{top}_{k}(s'') = \text{top}_{k}(s_{\psi''}) \) and observe that the requirements regarding computability of traversals by \( \text{pop}_{l-1} \) (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 \( \text{Graph}_{\ell}(\text{CPDA}(G)) \) 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 \( \text{Tr}(G) \) starting from the root (i.e. the trivial traversal \( \varepsilon \)). If \( s_m \) and \( t_m \) are labelled by the same element of \( \Sigma \cup \{ \varepsilon \} \), then the following conditions hold.

(i) Let \( u = \text{top}_1(s_m) \). 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 \( \text{top}_1(s_m) \) is the same as the final vertex in \( t_m \). Consequently, the respective definitions of traversals and \( \text{CPDA}(G) \) imply Property 7.

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

**Example 10.** Take the following traversal over the computation tree of \( G \) (recall that variable \( z \) has order 0 and variable \( \varphi \) has order 1) in Example 5 (see also Figure 2 for the associated HORS graph):

\[\text{We are abusing notation here. Technically, } \overline{\varphi} \text{ is a terminal symbol from the long transform } \overline{G}, \text{ hence it should have order 0 or 1. However, by the type of } \overline{\varphi} \text{ we mean } (\text{type}(\psi_1), \ldots, \text{type}(\psi_k), \psi), \text{ where } \overline{\varphi} = \psi_1 \cdots \psi_k.\]
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 \( \Lambda G \) rather than the exact symbol from \( V \).

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

\[
\hat{t} = \lambda \at \lambda z \at \lambda \varphi \at \varphi \lambda \varphi \lambda y \at g \lambda y g \lambda z
\]

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

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

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

To see the correspondence with the traversal \( t \), note that configurations \( s_2 \) and \( s_3 \) in Figures 6 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 \( t \) that end in node 7.
The traversal $t$ corresponding to $s_3$ is the prefix of $t$ that ends in the later occurrence of 7; we have

$$\hat{t} = \lambda@\lambda\Psi\Psi\lambda\varphi\zeta f \lambda \varphi \lambda \varphi x_1 x_2 \Psi \lambda \varphi \zeta f \lambda \varphi \lambda x'_1 x'_2 x_1$$

The reader might wish to check that $s_3 = \hat{t}$. (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 $\text{top}_1(s_m) = u$.

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.
Figure 6: A run of a 3-CPDA (Part 1 of 2).
Figure 7: A run of a 3-CPDA (Part 2 of 2).
For (i), observe that, because \( u \)'s label is not a variable, it follows from the definition of \( \text{CPDA}(G) \) that, if \( v \) was a \( j \)-child labelled by a lambda of type \( A \), then \( u \) would have to be labelled by a \( \Sigma \)-symbol and, thus, the order of \( A \) would be 0.

For (ii), \( t_m \xrightarrow{2m} t_{m+1} \) 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 \( \lambda_G(\text{top}_2(s_{m+1})) = \tau^1(t_m) \). We have \( \lambda_G(\text{top}_2(s_{m+1})) = \lambda_G(\text{top}_2(s_m))v \) and, in all three cases corresponding to the rules \((A), (S), (L)\), \( \tau^1(t_m) = \tau^1(t_m) v \) holds. Thus, by induction hypothesis, we get \( \lambda_G(\text{top}_2(s_{m+1})) = \tau^1(t_m) \). For (b), we note that \( s_{m+1} = s_m v \) and \( t_{m+1} = t_m v \). So, by induction hypothesis, \( s_{m+1} = t_{m+1} \). 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 \( \text{binder}(u) \) (note that then we have \( i \geq 1 \)) and suppose \( \text{binder}(u) \) 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 \( \varphi_i \) 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 \( \text{binder}(u) \) being a 0-child) that \( t \) has the following shape:

\[
\begin{array}{c}
t = \cdots \xrightarrow{0} u_0 \xrightarrow{1} \cdots \xrightarrow{u} \\
\@ \lambda \varphi_i
\end{array}
\]

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

Consider the operation \( \delta(u) = \text{push}_{n-l+1}; \text{pop}_1^{p+1}; \text{push}_1^{K_{\text{top}_1}(n-l+1)} \). By the induction hypothesis of (ii), the top 1-stack of \( s \) — call it \( \sigma \) — is the \( \text{P-view} \) of \( t \). Since the top 1-stack of \( \text{push}_{n-l+1} s \) is a copy of \( \sigma \), applying \( \text{push}_1^{p+1} \) to \( \text{push}_{n-l+1} s \) returns a stack that has the \( \@ \)-labelled node \( u_0 \) as the \( \text{top}_1 \)-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 \( \text{collapse}(s') = s \).

It follows from the structure of \( \lambda(G) \) that \( v \) must be labelled by \( \lambda \varphi_i \) (say) of the same type as the label \( \varphi_i \) 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 \( t \xrightarrow{2m} t' \). We shall show that \( s' \) computes \( t' \).

(a) We need to show \( \lambda_G(\text{top}_2(s')) = \tau^1 \). By definition of \( s' \), we have \( \lambda_G(\text{top}_2(s')) = \lambda_G(\text{top}_2(s) \leq u_0 v) \), i.e. \( \text{top}_2(s') \) is the prefix of the 1-stack \( \text{top}_2(s) \) — regarded as a sequence — up to and including the occurrence of \( u_0 \) described above, extended by \( v \). By induction hypothesis (a), we have \( \lambda_G(\text{top}_2(s)) = \tau^1 \). Thus

\[
\lambda_G(\text{top}_2(s')) = \tau^1 \leq u_0 v = \tau^1 u_0 v = \tau^1 v = \tau^1
\]

as required (the second equation holds because \( u_0 \) appears in \( \tau^1 \)).
(b) We have \( s' = sv \) (indeed \( v \) has a link and \( \text{collapse} \ s' = s \)) and \( \hat{v} = \hat{tv} \). Since \( s = \hat{t} \) by induction hypothesis (b), we have \( s' = \hat{v} \).

(c) Because the top 1-stack of \( s' \) is (a copy of) a prefix of \( \text{top}_2(s) \) 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 \( \text{top}_2(s') \) except (possibly) the single new link. Let \( \sigma' \) 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 \( \varphi_i \), which is the \( i \)th item of the list \( \overline{\varphi} \).

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

\[
\begin{array}{c}
\vdots \quad u_0 \quad u_1 \quad \ldots \quad u \quad \psi \quad \lambda \overline{\varphi} \quad \varphi_i \end{array}
\]

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

\[
s = \left[ \ldots , \ldots \sigma \ldots \left[ \ldots , \ldots , \ldots , u_1 \ldots u \right] \ldots \right]_{\text{top } 1-\text{stack of } s}
\]

\[
\left[ \ldots , \ldots , u_1 \ldots u \right]_{\text{top } (n-l)-\text{stack of } s, \text{i.e. } w}
\]

wherein \( u_1 \) has a link to some \((n-l)-\)stack \( \sigma \). Since \( l' \geq l \), the \((n-l)-\)stack \( \sigma \) is embedded in the top \((n-l)-\)stack of \( s \), as indicated by the figure above. Note that, by induction hypothesis (ii (d)), \( s_\sigma \) computes \( t_{<u_1} \). In particular the \( \text{top}_1-\)element of \( \sigma \) must be \( u_0 \).

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

\[
\text{pop}^2_{l}; \text{collapse}; \text{push}^1_{E_i(\text{top}_1), n-l+1},
\]

affects only the top (duplicate) copy of \( w \). Applying \( \text{pop}^2_{l} \) to \( \text{push}_{n-l+1}(s) \) returns a stack that has \( u_1 \) as the \( \text{top}_1 \)-element; the \( \text{collapse} \)-operation then reduces it to a stack that has a copy \( \sigma' \) (say) of \( \sigma \) as its top \((n-l')-\)stack, i.e. its \( \text{top}_1 \)-element is \( u_0 \). The node that is pushed 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 \( \lambda \overline{\varphi} \) (say) whose type is the same as that of \( \varphi_i \), 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 \( t \nearrow t' \). We need to show that \( s' \) computes \( t' \).

(a) Observe that \( \tau t' = \tau t_{<u_0} uv \). Since \( s_\sigma \) computes \( t_{<u_1} \) (by induction hypothesis (ii (d))), so does \( s'_{\sigma'} \). by Lemma \[2\] Hence, \( \lambda_G(\text{top}_2(s')) = \lambda_G(\text{top}_2(\sigma')) \). Hence, \( \lambda_G(\text{top}_2(\sigma')) \). Hence, \( \lambda_G(\text{top}_2(s')) = \lambda_G(\text{top}_2(\sigma')) \).

(b) Observe that \( s' = sv \) (indeed \( v \) has a link and \( \text{collapse}(s') = s \)) and \( \hat{v} = \hat{tv} \). Thus, by induction hypothesis, \( s' = \hat{v} \).
(c) Since \( s'_o \) computes \( t_{<u_1} \) and \( \text{top}_2(s') \) is a copy of \( \text{top}_2(s'_o) \) augmented with \( v \), (c) holds.

(d) We only need to verify (d) for the new link (all other links satisfy (d) because \( s'_o \) computes \( t_{<u_1} \)). Recall that \( v \) points at the stack \( w \). Since \( s'_w = s \) and \( t'_{<w} = t \), (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 \( \text{binder}(u) \) being a 0-child), \( t \) must have the following shape:

\[
t = \cdots u_0 \underbrace{u_1 \cdots u}_{\theta} \quad \text{at} \quad \lambda \varphi \quad x
\]

As in 1., \( \varphi \) has the shape \( \cdots u_0 \underbrace{u_1 \cdots u}_{\theta} \) and the segment \( \theta \) has length \( p + 1 \), where \( p \) is the span of the variable node \( u \).

Consider the operation \( \delta(u) = \text{pop}^{p+1}_1, \text{push}^E_{0}(\text{top}_i) \). Applying \( \text{pop}^{p+1}_1 \) to \( s \) returns a stack that has the \@-labelled node \( u_0 \) as the \( \text{top}_0 \)-element. The node that is \( \text{push}^0_1 \)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 \( \lambda \), 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 \( t \xrightarrow{i} t' \). We shall show that \( s' \) computes \( t' \).

(a) We need to show \( \lambda_G(\text{top}_2(s')) = \varphi \). By definition of \( s' \), we have \( \text{top}_2(s') = (\text{top}_2(s))_{\leq u_0} v \).

By induction hypothesis (ii), we have \( \lambda_G(\text{top}_2(s)) = \varphi \). Thus

\[
\lambda_G(\text{top}_2(s')) = \varphi\quad \text{as required.}
\]

The second equation follows from the definition of \( \varphi \) and the fact that \( u_0 \) appears in \( \varphi \).

(b) We have \( s' = (\text{pop}^{p+1}_1(s)) v \) and \( \hat{t} = t_{\leq u_0} v \). By induction hypothesis (ii (c)), \( \text{pop}^{p+1}_1(s) \) computes \( t_{\leq u_0} \), in particular \( \text{pop}^{p+1}_1(s) = t_{\leq u_0} \). 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 \( i \)-th item of the list \( \varphi \).

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

\[
t = \cdots u_0 \underbrace{u_1 \cdots u}_{\psi} \quad \text{at} \quad \lambda \varphi \quad x
\]
Further, the variable $\psi$ has the same type as $\lambda x\varphi$, which (say) is of order $l'$. It follows that $l' > l$. By induction hypothesis (i) and (ii), $s$ has the following shape

$$s = [\cdots \sigma \cdots [\cdots u_1 \cdots u] \cdots]$$

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

Now, to understand what $s'$ looks like, consider the operation $\delta(u) = pop_1^p; \text{collapse}; push_{E_1(top_1)}^{E_1(top_1)}$. 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 $\sigma$ as its top $(n - l')$-stack, i.e. its $top_1$-element is $u_0$. The node that is then $push_{E_1}$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 $\lambda$. 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 $t \xrightarrow{\lambda} t'$. We need to show that $s'$ computes $t'$.

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

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

(c) Because $s_{\sigma}$ computes $t_{<u_1}$, condition (c) holds.

(d) Again, it suffices to appeal to the fact that $s_{\sigma}$ 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 $\mu$-calculus formula $\varphi$ and a transition graph $Graph(A)$ of a CPDA. Deciding whether $\varphi$ holds in some vertex $v$ of the graph is equivalent to decide whether the same formula $\varphi$ is true at the root of the tree obtained by unfolding $Graph(A)$ 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 $\mu$-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 [33] imply that $\mu$-calculus model-checking is decidable for transition graphs of CPDA.

As $\mu$-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/\(\mu\)-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 \(\mu\)-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 \((\text{RecTree}_n \Sigma)_n\) 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 Cauca definition for the \(\varepsilon\)-closure of CPDA graphs? As trees generated

\(^{12}\)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 \(R_{\varphi}\) is a \(\varphi\)-reflection of \(R\) just if

- \(R\) and \(R_{\varphi}\) 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 \(f\) if \(u\) in \([R]\) satisfies \(\varphi\); it is \(f\) otherwise.

Thus if \([R]\) is the computation tree of a program \(R\), we may regard \(R_{\varphi}\) as a transform of \(R\) that can internally observe its behaviour against a specification \(\varphi\). We say that \(\mathcal{R}\) is (constructively) reflective w.r.t. \(\mathcal{L}\) just if there is an algorithm that transforms a given pair \((R, \varphi)\) to \(R_{\varphi}\).
by \(n\)-CPDA are exactly those obtained by unravelling and unfolding an \(n\)-CPDA graph, is there a class of transformations \(T\) from trees to graphs such that every \((n+1)\)-CPDA graph is obtained by applying a \(T\)-transformation to some tree generated by an \(n\)-CPDA. Note that a \(T\)-transformation may in general not preserve MSO decidability (as \(n\)-CPDA graphs have undecidable MSO theory [34]), but should preserve modal \(\mu\)-calculus decidability of trees generated by \(n\)-CPDA.

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.

References


[66] Gérard Sénizergues. The equivalence problem for deterministic pushdown automata is decid-
Collapsible Pushdown Automata and Recursion Schemes

M. Hague, A. S. Murawski, C.-H. L. Ong, O. Serre


