Recursion Schemes, Collapsible Pushdown Automata and Higher-Order Model Checking

Luke Ong

University of Oxford, UK

Abstract. This paper is about two models of computation that underpin recent developments in the algorithmic verification of higher-order computation. Recursion schemes are in essence the simply-typed lambda calculus with recursion, generated from first-order symbols. Collapsible pushdown automata are a generalisation of pushdown automata to higher-order stacks — which are iterations of stack of stacks — that contain symbols equipped with links. We study and compare the expressive power of the two models and the algorithmic properties of infinite structures such as trees and graphs generated by them. We conclude with a brief overview of recent applications to the model checking of higher-order functional programs. A central theme of the work is the fruitful interplay of ideas between the neighbouring fields of semantics and algorithmic verification.

1 Introduction

Over the past decade, there has been significant progress in the development of finite-state and pushdown model checking for software verification. Though highly effective when applied to first-order imperative programs such as C, these techniques are less useful for higher-order functional programs. In contrast, the standard approaches to the verification of higher-order programs are *type-based static analysis* on the one hand, and *theorem proving* and *dependent types* on the other. The former is sound but imprecise; the latter typically requires human intervention.

Recently an approach to model checking higher-order programs based on *recursion schemes* has emerged as a verification methodology that promises to combine accurate analysis with push-button automation. A grammar for generating possibly-infinite trees, recursion schemes are in essence the simply-typed λ-calculus with recursion, built up from first-order symbols. Ong [53] proved that the trees generated by recursion schemes have decidable monadic second-order (MSO) theories, subsuming earlier well-known MSO decidability results for regular (order-0) [63] and algebraic (order-1) trees [15]. Building on [53], Kobayashi [39] introduced a novel approach to the verification of higher-order functional programs by reduction to the *recursion schemes model checking problem*: does the tree generated by a given recursion schemes satisfy a given correctness property?

This survey paper concerns two models of higher-order computation: recursion schemes and collapsible pushdown automata. Recursive program schemes are an old formalism for the semantical analysis of both imperative and functional programs [52, 19, 14]. Recursion schemes, which generalise recursive program schemes to higher orders, are a compelling model of higher-order functional programs. Pushdown automata characterise the control flow of first-order recursive programs [34]; pushdown model checkers (such as Moped [26]) are an important component of state-of-the-art software model checkers. The two models are suitable for the algorithmic verification of higher-order computation. On the one hand, they model higher-order computation accurately: recursion schemes are a version of PCF [62]; and collapsible pushdown automata compute exactly the innocent strategies, which give rise to the fully abstract model of PCF [32, 49, 29]. On the other, they enjoy rich algorithmic properties [36, 12, 53, 29].

The goal of our work is to use semantic methods, in conjunction with algorithmic ideas and techniques from verification, to formally analyse programming situations in which higher-order features are important. In Section 2, we present two families of generators of infinite structures: recursion schemes and higher-order pushdown automata; we study their relationship and characterise their expressivity. In Section 3, we survey recent results on the model checking of trees generated by recursion schemes. In Section 4, we introduce collapsible pushdown automata, study their relationship with recursion schemes, and discuss developments in the solution of parity games over the configuration graphs of collapsible pushdown systems. In Section 5, we briefly discuss recent applications to the model checking of higher-order functional programs, and then conclude.

2 Two Families of Generators of Infinite Structures

2.1 Higher-Order Pushdown Automata

Higher-order pushdown automata were introduced by Maslov [45, 46] as a generalisation of pushdown automata and nested pushdown automata. Let Γ be a stack alphabet that contains a distinguished *bottom-of-stack symbol* $\bot$. An *order-0 stack* is just a stack symbol. An *order-*$(n+1)$ *stack* is a non-null sequence (written $[s_1 \cdots s_l]$) of order-n stacks. We often abbreviate order-n stack to n-stack, and write $n\text{-}Stack_\Gamma$ for the set of n-stacks over Γ. As usual, $\bot$ cannot be popped from or pushed onto a stack. (Thus we require an *order-1 stack* to be a non-null sequence $[a_1 \cdots a_l]$ of Γ-symbols such that for all $1 \leq i \leq l$, $a_i = \bot$ if and only if $i = 1$.) We define $\bot_k$, the *empty k-stack*: $\bot_0 := \bot$ and $\bot_{k+1} := [\bot_k]$. When displaying examples of n-stacks, we shall omit $\bot$ to avoid clutter.

Operations on n-Stacks The following operations are defined on 1-stacks:

$$\begin{array}{rll} push_1^Z\ [Z_1 \cdots Z_{i-1}\ Z_i] & := [Z_1 \cdots Z_{i-1},\ Z_i, Z] & \text{where } Z \in \Gamma \setminus \{\bot\} \\ pop_1\ [Z_1 \cdots Z_{i-1}\ Z_i] & := [Z_1 \cdots Z_{i-1}] & \text{where } Z_i \neq \bot \\ top_1\ [Z_1 \cdots Z_{i-1}\ Z_i] & := Z_i & \end{array}$$

The following operations are defined on n-stacks, where $n \geq 2$:

$$\begin{aligned}
push_n\,[s_1 \cdots s_{i-1}\,s_i] &:= [s_1 \cdots s_{i-1}\,s_i\,s_i] \\
pop_n\,[s_1 \cdots s_{i-1}\,s_i] &:= [s_1 \cdots s_{i-1}] \\
push_k\,[s_1 \cdots s_{i-1}\,s_i] &:= [s_1 \cdots s_{i-1} push_k\,s_i] \quad \text{where } 2 \leq k < n \\
push_1^Z\,[s_1 \cdots s_{i-1}\,s_i] &:= [s_1 \cdots s_{i-1}\,push_1^Z\,s_i] \quad \text{where } Z \in \Gamma \setminus \{\,\bot\,\} \\
pop_k\,[s_1 \cdots s_{i-1}\,s_i] &:= [s_1 \cdots s_{i-1}\,pop_k\,s_i] \quad \text{where } 1 \leq k < n \\
top_n\,[s_1 \cdots s_{i-1}\,s_i] &:= s_i \\
top_k\,[s_1 \cdots s_{i-1}\,s_i] &:= top_k\,s_i \quad \text{where } 1 \leq k < n
\end{aligned}$$

For $1 \leq k \leq n$, the operation pop_k is undefined on any n-stack such that its top k-stack (or the n-stack itself, in case $k = n$) has only one element. For example $pop_2\,[[\bot\,\alpha\,\beta]]$ and $pop_1\,[[\bot\,\alpha\,\beta]\,[\bot]]$ are both undefined. For $n \geq 0$, we define the set Op_n of *order-n stack operations*:

$$\begin{aligned}
Op_1 &:= \{\, push_1^Z \mid Z \in \Gamma \setminus \{\,\bot\,\}\,\} \cup \{\, pop_1\,\} \\
n \geq 2,\ Op_n &:= \{\, push_k, pop_k \mid 2 \leq k \leq n\,\} \cup Op_1.
\end{aligned}$$

Higher-order pushdown automata were first used to define word languages. Here we take a somewhat generic approach to the definition.

Definition 1. An *abstract store system* is a tuple $\langle \Gamma, AStore_\Gamma, Op, top, \bot \rangle$ where Γ is a (finite) store alphabet, $AStore_\Gamma$ is a set of *abstract stores* notionally generated from Γ, Op is a set of *abstract store operations* which are just partial functions $AStore_\Gamma \rightharpoonup AStore_\Gamma$, $top : AStore_\Gamma \to \Gamma$ is an *abstract read* function, and $\bot \in AStore_\Gamma$ is the initial abstract store.

For example, for $n \geq 0$, the tuple $\langle \Gamma, n\text{-}Stack_\Gamma, Op_n, top_1, \bot_n \rangle$, which we shall call the *system of order-n stacks over* Γ, is an abstract store system. Another example is the *system of order-n collapsible stacks*, which will be introduced in Section 4.1. The semi-infinite tape (of a Turing machine) and the FIFO queue (of a Minsky machine) are also examples of abstract store system.

Definition 2. (i) Let $\mathcal{S} = \langle \Gamma, AStore_\Gamma, Op, top, \bot \rangle$ be an abstract store system. A *word-language* $\mathcal{S}$*-automaton* is a tuple $\mathcal{A} = \langle \mathcal{S}, Q, \Sigma, \Delta, q_I, F \rangle$ where Q is a finite set of states, Σ is an input alphabet, $\Delta \subseteq Q \times (\Sigma \cup \{\,\epsilon\,\}) \times \Gamma \times Op \times Q$ is a transition relation, $q_I \in Q$ is the initial state, and $F \subseteq Q$ is a set of final states. A *configuration* is a pair (q, s) where $q \in Q$ and $s \in AStore_\Gamma$; the initial configuration is $(q_I, \bot)$ where $\bot \in AStore_\Gamma$. The transition relation Δ induces a transition relation between configurations according to the rule: if $(q, a, Z, \theta, q') \in \Delta$ and $top(s) = Z$ then $(q, s) \xrightarrow{a} (q', \theta(s))$. A word $w \in \Sigma^*$ is *accepted* by $\mathcal{A}$ just in case there is a sequence of transitions of the form $(q_I, \bot) \xrightarrow{a_1} (q_1, s_1) \xrightarrow{a_2} \cdots \xrightarrow{a_m} (q_m, s_m)$ such that $q_m \in F$, $s_m = \bot$ and $w = a_1 \cdots a_m$.

(ii) We say that a word-language $\mathcal{S}$-automaton is *deterministic* just if Δ is a partial function $Q \times (\Sigma \cup \{\,\epsilon\,\}) \times \Gamma \rightharpoonup Op \times Q$; further, for every q and Z, if Δ is defined on (q, ϵ, Z), then it must be undefined on (q, a, Z) for every $a \in \Sigma$.

(iii) In case $\mathcal{S} = \langle \Gamma, n\text{-}Stack_\Gamma, Op_n, top_1, \bot_n \rangle$ is the system of n-stacks over Γ, we refer to a word-language $\mathcal{S}$-automaton as an *order-n pushdown word-language automaton*, and specify it as $\langle \Gamma, Q, \Sigma, \Delta, q_I, F \rangle$. When it is clear from the context, we call it an order-n pushdown automaton or simply n-PDA.

By definition, order-0 PDA are finite-state automata. Order-1 PDA are (ordinary) PDA. Order-2 PDA capture the *indexed languages* of Aho [4]. There are several major studies on higher-order PDA in the literature [19, 25]. [1]

Example 3. (i) The language $L = \{ a^n b^n c^n \mid n \geq 0 \}$ is recognisable by an order-2 deterministic PDA $\langle \{ \bot, Z \}, \{ q_1, q_2, q_3 \}, \{ a, b, c \}, \delta, q_1, \{ q_1, q_3 \} \rangle$ where $\delta : Q \times (\Sigma \cup \{ \epsilon \}) \times \Gamma \rightharpoonup Op_2^* \times Q$ is defined as follows:

$$\begin{array}{lll} (q_1, a, \bot) \mapsto (push_2 \,;\, push_1^Z, q_1) & (q_1, b, Z) \mapsto (pop_1, q_2) & (q_2, c, \bot) \mapsto (pop_2, q_3) \\ (q_1, a, Z) \mapsto (push_2 \,;\, push_1^Z, q_1) & (q_2, b, Z) \mapsto (pop_1, q_2) & (q_3, c, Z) \mapsto (pop_2, q_3) \end{array}$$

For example, the following computation accepts $a^2\, b^2\, c^2$.

$$\begin{array}{ccccc} q_1\,[[]] & \xrightarrow{a} & q_1\,[[]\,[Z]] & \xrightarrow{a} & q_1\,[[]\,[Z]\,[ZZ]] \\ & & & & \downarrow b \\ & & & & q_2\,[[]\,[Z]\,[Z]] \\ & & & & \downarrow b \\ q_3\,[[]] & \xleftarrow[c]{} & q_3\,[[]\,[Z]] & \xleftarrow[c]{} & q_2\,[[]\,[Z]\,[]] \end{array}$$

(ii) Using the same idea, we can use order-2 deterministic pushdown automata to recognise $\{ a^n\, b^n\, c^n\, d^n \mid n \geq 0 \}$ and so on.

Several basic properties of the pushdown hierarchy of word languages were proved by Maslov [45, 46].

Theorem 4 (Maslov 1974). *Let $n \geq 0$.*

(i) The emptiness problem for order-n PDA is decidable.

(ii) The order-n languages form an abstract family of languages[2]*.*

(iii) Higher-order PDA define an infinite hierarchy of word languages.

In the sequel, we shall use higher-order pushdown automata and a new variant, called *collapsible pushdown automata*, to generate infinite structures such as languages of infinite trees (Section 4.1) and infinite graphs (Section 4.2). These structures have rich algorithmic properties.

[1] Engelfriet's work used a somewhat different (but equivalent) machine called *iterated pushdown automata*.

[2] An *abstract family of languages* (AFL) is a collection of languages closed under union, concatenation, Kleene star, intersection with regular languages, homomorphism and inverse homomorphism.

2.2 Recursion Schemes

Recursion schemes are a method of constructing possibly infinite term-trees (or sets of such trees). The idea goes back to Park's pioneering work on program schemes and fixpoint theory [57] in the late 1960s and Patterson's PhD thesis [61]. Program schemes are a program calculus that clearly separates control structure and operations on data, thus providing a framework for investigating purely structural program transformation and the descriptive power of control structures. There is a large literature [5, 51, 52, 28]throughout the 1970s and much of the 1980s on the semantics and transformation of recursive program schemes. In the late 1970s Damm, Fehr and Indermark [18, 20, 19] introduced program schemes constrained by a family of simple types, and considered them as generators of finite-word and tree languages. For a survey of the early work, see Courcelle's handbook article [14].

For us, *types* are simple types defined by the grammar $A ::= o \mid A \to B$. By convention, arrows associate to the right; thus every type can be written uniquely as $A_1 \to \cdots \to A_n \to o$, for some $n \geq 0$. We define the *order* of a type: $ord(o) := o$ and $ord(A \to B) := \max(ord(A) + 1, ord(B))$. Intuitively the order of a type measures how deeply nested it is on the left of the arrow.

Assume a countably infinite set Var of typed variables. Let Θ be a set of typed symbols such as Var; we write $s : A$ to mean s has type A. The set of *applicative terms* generated from Θ is the least set containing Θ closed under the *application rule*: if $s : A \to B$ and $t : A$ then $s\,t : B$. By convention, application associates to the left. Given a term $s : A$, we define $ord(s) := ord(A)$.

Definition 5. A *higher-order recursion scheme* (or simply *recursion scheme*) is a tuple $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where

- Σ is a *ranked alphabet of terminals* i.e. each $f \in \Sigma$ has an arity $\mathsf{ar}(f) \geq 0$, which is written $f : \mathsf{ar}(f)$ by abuse of notation; we assume that f has the type $\underbrace{o \to \cdots \to o}_{\mathsf{ar}(f)} \to o$.
- $\mathcal{N}$ is a set of typed *non-terminals*; $S \in \mathcal{N}$ is a distinguished *start symbol* of type o.
- $\mathcal{R}$ is a finite set of rewrite rules of the form $F\,\xi_1 \cdots \xi_n \to e$ where $F : A_1 \to \cdots \to A_n \to o$, each $\xi_i : A_i$, and $e : o$ is an applicative term generated from $\Sigma \cup \mathcal{N} \cup \{\xi_1, \cdots, \xi_n\}$.

The *order* of a recursion scheme is defined to be the highest order (of the type) of its non-terminals. In the following we shall use uppercase letters F, G, H, etc. to range over non-terminals, lowercase letters f, g, h, etc. to range over terminals, and $\xi, \varphi, \psi, x, y, z$, etc. to range over variables.

We use *deterministic* recursion schemes (i.e. at most one rewrite rule for each non-terminal) to define possibly-infinite trees. (In general, recursion schemes define tree languages. Henceforth, when defining trees, we assume that recursion schemes are deterministic.) Given a recursion scheme G, the term-tree generated

by G, denoted $[\![G]\!]$, is the possibly-infinite applicative term *constructed from the terminals*, which is obtained from the start symbol S by unfolding the rewrite rules *ad infinitum*, replacing formal by actual parameters each time.

Example 6 (An order-1 recursion scheme G_1). Take the ranked alphabet $\Sigma = \{ f : 2, g : 1, a : 0 \}$. Consider the order-1 recursion scheme G_1 with rewrite rules:

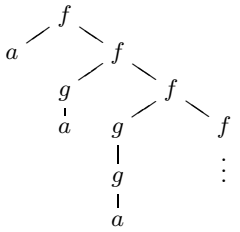
$$S \to F\, a \qquad F\, x \to f\, x\, (F\, (g\, x))$$

where $F : o \to o$. Unfolding from the start symbol S, we have

$$S \to F \to f\, a\, (F\, (g\, a)) \to f\, a\, (f\, (g\, a)\, (F\, (g\, (g\, a)))) \to \cdots$$

thus generating the infinite applicative term

$$f\, a\, (f\, (g\, a)\, (f\, (g\, (g\, a))(\cdots))).$$



Because the rewrite system is Church-Rosser, it does not matter which reduction strategy is used provided it is *fair* in the sense of not neglecting any branch. The tree generated by G_1, $[\![G_1]\!]$, is the abstract syntax tree of the infinite term as shown on the right.

Formally the rewrite relation $\to_G$ is defined by induction over the rules:

$$\frac{F\xi_1 \cdots \xi_n \to e \text{ is a } \mathcal{R}\text{-rule}}{Ft_1 \cdots t_n \to_G e[t_1/\xi_1, \cdots, t_n/\xi_n]} \qquad \frac{t \to_G t'}{s\, t \to_G s\, t'} \qquad \frac{t \to_G t'}{t\, s \to_G t'\, s}$$

We write $\to_G^*$ for the reflexive, transitive closure of $\to_G$.

Let $l := \max \{ \mathsf{ar}(f) \mid f \in \Sigma \}$. A *$\Sigma$-labelled tree* is a partial function t from $\{ 1, \cdots, l \}^*$ to Σ such that $dom(t)$ is prefix-closed; we assume that t is *ranked* i.e. if $t(w) = a$ and $\mathsf{ar}(a) = m$ then $\{ i \mid w\, i \in dom(t) \} = \{ 1, \cdots, m \}$. A (possibly infinite) sequence π over $\{ 1, \cdots, l \}$ is a *path* of t if every finite prefix of π is in $dom(t)$. Given a term t, we define a (finite) tree $t^\perp$ by:

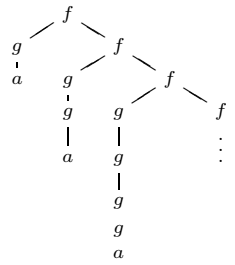
$$t^\perp := \begin{cases} f & \text{if } t \text{ is a terminal } f \\ {t_1}^\perp {t_2}^\perp & \text{if } t \text{ is of the form } t_1 t_2 \text{ and } {t_1}^\perp \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

For example $(f\ (F\ a)\ b)^\perp = f\ \perp\ b$. Let $\sqsubseteq$ be the partial order on $\Sigma \cup \{ \perp \}$ defined by $\forall a \in \Sigma. \perp \sqsubseteq a$. We extend $\sqsubseteq$ to a partial order on trees by: $t \sqsubseteq s := \forall w \in dom(t). (w \in dom(s) \wedge t(w) \sqsubseteq s(w))$. For example, $\perp \sqsubseteq f\ \perp\ \perp \sqsubseteq f\ \perp\ b \sqsubseteq f\ a\ b$. For a directed set T of trees, we write $\bigsqcup T$ for the least upper bound of elements of T with respect to $\sqsubseteq$. We define the *tree generated by G*, or the *value tree* of G, by $[\![G]\!] := \bigsqcup \{ {t}^\perp \mid S \to_G^* t \}$. By construction, $[\![G]\!]$ is a possibly infinite, ranked $(\Sigma \cup \{ \perp \})$-labelled tree. For $n \geq 0$, we write $RecSchTree_n^\Sigma$ for the class of Σ-labelled trees generated by order-n recursion schemes.

Example 7 (An order-2 recursion scheme G_2).

Take $\Sigma = \{ f : 2, g : 1, a : 0 \}$, with rewrite rules:

$$\begin{aligned} S &\to F\, g \\ B\, \varphi\, \psi\, x &\to \varphi\, (\psi\, x) \\ F\, \varphi &\to f\, (\varphi\, a)\, (F\, (B\, \varphi\, \varphi)) \end{aligned}$$



where $B : (o \to o) \to (o \to o) \to o \to o$ and $F : (o \to o) \to o$. The value tree, $[\![G_2]\!] : \{ 1, 2 \}^* \longrightarrow \Sigma$, is shown on the right.

Recursion Schemes as Generators of Word Languages By viewing a word as a linear tree i.e. a tree with branching factor at most one, we can use (non-deterministic) recursion schemes to generate word languages. Thus a finite word "$a\, b\, c$" (say) is represented by the applicative term $a\, (b\, (c\, e))$, where a, b and c are now regarded as terminal symbols of arity 1, and e is a distinguished nullary end-of-word marker. We call such a recursion scheme a *word-language recursion scheme*.

Example 8. (i) The language $\{ a^n\, b^n \mid n \geq 0 \}$ is generated by the order-1 recursion scheme: $S \to F\, e \quad F\, x \to a\, (F\, (b\, x)) \quad F\, x \to x$

(ii) The language $\{ a^n\, b^n\, c^n \mid n \geq 0 \}$ is generated by the order-2 recursion scheme:

$$\begin{aligned} &S \to F\, I\, e && F\, \varphi\, x \to F\, (H\, \varphi)\, (c\, x) && I\, x \to x \\ &F\, \varphi\, x \to \varphi\, x && H\, \varphi\, y \to a\, (\varphi\, (b\, y)) \end{aligned}$$

where $F : (o \to o) \to o \to o$, $H : (o \to o) \to o \to o$ and $I : o \to o$.

Take a rewrite rule $F\, \overline{x} \to s$ for the non-terminal F. If the head symbol of s is a terminal symbol a, then we say that it is an (F, a)-rule; otherwise it is an (F, ϵ)-rule. We say that a word-language recursion scheme is *deterministic* just if for every non-terminal F and $\xi \in \Sigma \cup \{ \epsilon \}$, there is at most one (F, ξ)-rule; further, for each F, if there is a (F, ϵ)-rule, then there can be no (F, a)-rule where $a \in \Sigma$. Thus both schemes in Example 8 are non-deterministic. However there are deterministic recursion schemes that generate the languages of the Example.

Example 9. The following deterministic recursion scheme generates the language $\{ a^n\, b^n\, c^n \mid n \geq 0 \}$.

$$\begin{aligned} &S \to e && A\, \varphi_1\, \varphi_2 \to b\, (\varphi_1\, (\varphi_2\, e)) \\ &S \to a\, (A\, I\, (F\, c\, I)) && F\, \varphi_1\, \varphi_2\, x \to \varphi_1 (\varphi_2\, x) \\ &A\, \varphi_1\, \varphi_2 \to a\, (A\, (F\, b\, \varphi_1)\, (F\, c\, \varphi_2)) && I\, x \to x \end{aligned}$$

where $A : (o \to o) \to (o \to o) \to o$, $F : (o \to o) \to (o \to o) \to o \to o$ and $I : o \to o$.

2.3 Maslov's Pushdown Hierarchy of Word Languages

In the original 1974 paper [45], Maslov mentioned in brief several formalisms that are equivalent to higher-order pushdown automata. In a subsequent paper in 1976 [46], he set out the details of one such formalism, called *higher-order indexed grammars*. Using a key operation of "raising a language to a power given by another language", higher-order indexed grammars generalise Aho's indexed grammars [4] (which are the order-2 indexed grammars) to all finite orders; further, infinite-order indexed grammars define exactly the recursively enumerable languages. In a different direction, Damm [18, 19] studied a system of recursion schemes that are constrained by *derived types*. Damm and Geordt [19, 21] showed that, as generators of word languages, order-n safe recursion schemes are equivalent to order-n pushdown automata, which are in turn equivalent to order-n indexed grammars. The notion of safety is introduced in Section 2.4.

Theorem 10 (Maslov 1976, Damm 1982, Damm and Goerdt 1986). *For each $n \geq 0$, the following formalisms define the same class of word languages:*

(i) order-n pushdown automata
(ii) order-n indexed grammars
(iii) order-n safe recursion schemes.

The low-order languages of Maslov's pushdown hierarchy are well-known and much studied. The order-2 languages, which are indexed languages, were a topic of interest in the 1970s and early 1980s [30, 27, 56, 23]. A notable recent advance [60] was Parys' pumping lemma for the class of ϵ-closure of order-n pushdown graphs, for every $n \geq 0$. (The result was subsequently extended by Kartzow and Parys [35] to a pumping lemma for the hierarchy of ϵ-closure of collapsible pushdown graphs.) Another significant development was the result, due to Inaba and Maneth at FSTTCS 2008 [33], that every language in the pushdown hierarchy (equivalently the OI hierarchy [19]) is context sensitive. By considering word languages in the image of iterated composites of macro tree transducers, they show that languages of the pushdown hierarchy are in non-deterministic linear space and NP-complete [33, Corollary 9].

An interesting and challenging problem is to find higher-order analogues of Parikh's Lemma and such powerful characterisation results as the Myhill-Nerode Theorem. There is a famous logical characterisation of regular languages due to Büchi: the order-0 languages are exactly those definable by S1S, monadic second-order logic with one-successor. Lautemann et al. [44] extended the characterisation to context-free languages by augmenting S1S with a notion of quantification over matchings. To our knowledge, no such logical characterisations are known for languages of order 2 or higher.

Much of what is known about the complexity of languages recognisable by automata with higher-order stacks (and other auxiliary memory) is due to Engelfriet. His seminal paper [25] contains a wealth of results. We pick out a few here that are particularly useful for studying the algorithmics of infinite structures generated by higher-order PDA.

Theorem 11 (Engelfriet 1991). *Let $s(n) \geq \log(n)$.*

(i) For $k \geq 0$, the word acceptance problem of non-deterministic order-k PDA with a two-way work-tape with $s(n)$ space is k-EXPTIME complete.
(ii) For $k \geq 1$, the word acceptance problem of alternating order-k PDA with a two-way work-tape with $s(n)$ space is $(k-1)$-EXPTIME complete.
(iii) For $k \geq 0$, the word acceptance problem of alternating order-k PDA is k-EXPTIME complete.
(iv) For $k \geq 1$, the emptiness problem of non-deterministic order-k PDA is $(k-1)$-EXPTIME complete.

2.4 The Hierarchy of Higher-Order Pushdown Trees

Fix a ranked alphabet Σ with $m := \max\{\, \mathsf{ar}(f) \mid f \in \Sigma \,\}$. The branch language [13] of a Σ-labelled ranked tree t is a set of finite and infinite words that represent its maximal branches (or paths). Writing $[m] = \{\, 1, \cdots, m \,\}$, the *branch language* of $t : dom(t) \to \Sigma$ consists of:

- infinite words $(f_1, d_1)(f_2, d_2) \cdots$ such that there exists $d_1\, d_2 \cdots \in [m]^\omega$ with $t(d_1 \cdots d_i) = f_{i+1}$ and $d_{i+1} \leq \mathsf{ar}(f_{i+1})$ for every $i \geq 0$, and
- finite words $(f_1, d_1) \cdots (f_n, d_n)\, f_{n+1}$ such that there exists $d_1 \cdots d_n \in [m]^*$ with $t(d_1 \cdots d_i) = f_{i+1}$ for every $0 \leq i \leq n$, $d_i \leq \mathsf{ar}(f_i)$ for every $1 \leq i \leq n$, and $\mathsf{ar}(f_{n+1}) = 0$.

For example, the branch language of the tree generated by the recursion scheme G_1 of Example 6 is $\{\, (f,2)^\omega \,\} \cup \{\, (f,2)^n\, (f,1)\, (g,1)^n\, a \mid n \geq 0 \,\}$. It follows from the definition that two ranked trees are equal if and only if they have the same branch language.

Definition 12. (i) Let $\mathcal{S} = \langle \Gamma, AStore_\Gamma, Op, top, \bot \rangle$ be an abstract store system. A *tree $\mathcal{S}$-automaton* is a tuple $\mathcal{A} = \langle \mathcal{S}, Q, \Sigma, \delta, q_I \rangle$ where Q is a finite set of states, $q_I \in Q$ is the initial state, Σ is a ranked alphabet, and

$$\delta : Q \times \Gamma \longrightarrow (Q \times Op \cup \{\, (f, q_1 \cdots q_{\mathsf{ar}(f)}) \mid f \in \Sigma, q_i \in Q \,\})$$

is a transition function. A *configuration* is either a pair (q, s) where $q \in Q$ and $s \in AStore_\Gamma$, or a triple of the form $(f, q_1 \cdots q_{\mathsf{ar}(f)}, s)$ where $f \in \Sigma$ and $q_1 \cdots q_{\mathsf{ar}(f)} \in Q^*$. Let $\overline{\Sigma}$ be the label-set $\{\, (f,i) \mid f \in \Sigma, 1 \leq i \leq \mathsf{ar}(f) \,\} \cup \{\, a \in \Sigma \mid \mathsf{ar}(a) = 0 \,\}$. We define the labelled transition relation between configurations induced by δ:

$$\begin{array}{ll}
(q,s) \xrightarrow{\epsilon} (q', \theta(s)) & \text{if } \delta(q, top\, s) = (q', \theta) \\
(q,s) \xrightarrow{\epsilon} (f, \overline{q}, s) & \text{if } \delta(q, top\, s) = (f, \overline{q}) \text{ and } \mathsf{ar}(f) \geq 1 \\
(q,s) \xrightarrow{a} (a, \epsilon, s) & \text{if } \delta(q, top\, s) = (a, \epsilon) \text{ and } \mathsf{ar}(a) = 0 \\
(f, \overline{q}, s) \xrightarrow{(f,i)} (q_i, s) & \text{for every } 1 \leq i \leq \mathsf{ar}(f)
\end{array}$$

Let w be a finite or infinite word over the alphabet $\overline{\Sigma}$. We say that w is a *trace* of $\mathcal{A}$ just if there is a possibly-infinite sequence of transitions $(q_I, \bot) \xrightarrow{\ell_1} \gamma_1 \cdots \xrightarrow{\ell_m} \gamma_m \xrightarrow{\ell_{m+1}} \cdots$ such that $w = \ell_1 \ell_2 \cdots$. We say that $\mathcal{A}$ *generates* the Σ-labelled tree t just in case the branch language of t coincides with the set of traces of $\mathcal{A}$.

(ii) In case $\mathcal{S} = \langle \Gamma, n\text{-}Stack_\Gamma, Op_n, top_1, \bot_n \rangle$ is the system of n-stacks over Γ, we refer to a tree $\mathcal{S}$-automaton as an *order-n pushdown tree automaton*, and simply specify it by the tuple $\langle \Gamma, Q, \Sigma, \delta, q_I \rangle$. For $n \geq 0$, we write $PushdownTree_n^\Sigma$ for the class of Σ-labelled ranked trees generated by order-n tree pushdown automata.

Example 13. The tree of Example 6 is generated by the order-1 pushdown tree automaton $\langle \{ Z, \bot \}, \{ q_I, q_1, q_2 \}, \Sigma, \delta, q_I \rangle$ where δ is defined as follows:

$$(q_I, \bot) \mapsto (f, q_1\, q_2) \quad (q_2, \bot) \mapsto push_1^Z ; (f, q_1\, q_2) \quad (q_2, Z) \mapsto push_1^Z ; (f, q_1\, q_2)$$
$$(q_1, \bot) \mapsto (a, \epsilon) \qquad (q_1, Z) \mapsto pop_1 ; (g, q_1).$$

In a FoSSaCS 2002 paper [36], Knapik, Niwiński and Urzyczyn introduced the class $SafeRecTree_n^\Sigma$ of Σ-labelled trees generated by order-n recursion schemes that are *homogeneously typed* and satisfy a syntactic constraint called *safety*. They proved that for every $n \geq 0$, $SafeRecTree_n^\Sigma = PushdownTree_n^\Sigma$. Thus $SafeRecTree_0\Sigma$, the order-0 trees, are the regular trees (i.e. trees generated by finite-state transducers), and $SafeRecTree_1\Sigma$, the order-1 trees, are those generated by order-1 pushdown tree automata.

Later in the year, in an MFCS 2002 paper [12], Caucal introduced a hierarchy of trees and a dual hierarchy of graphs, which are defined by mutual recursion, using a pair of powerful functors.[3] The functor from graphs to trees is the unravelling operation, and the functor from trees to graphs is given by inverse rational mapping. Level 0 of the graph hierarchy are the finite. Caucal showed [12, Theorem 3.5] that $SafeRecTree_n^\Sigma = CaucalTree_n^\Sigma$, where $CaucalTree_n^\Sigma$ consists of Σ-labelled trees obtained from the regular Σ-labelled-trees (i.e. trees from $PushdownTree_0^\Sigma$) by iterating n-times the operation of inverse deterministic rational mapping followed by unravelling. To summarise:

Theorem 14. *For all $n \geq 0$, $SafeRecTree_n^\Sigma = PushdownTree_n^\Sigma = CaucalTree_n^\Sigma$.*

2.5 Homogeneous Types and the Safety Constraint

Safety [36] is a syntactic constraint on the rewrite rules of a recursion scheme. It specifies whether a formal parameter of a rewrite rule may occur in a subterm of the RHS of the rule, depending on the position of the subterm, and the respective orders of the parameter and the subterm.

Definition 15. (i) A type $A_1 \to \cdots \to A_n \to o$ is *homogeneous* just if each A_i is homogeneous, and $ord(A_1) \geq ord(A_2) \geq \cdots \geq ord(A_n)$. (It follows that the base type o is homogeneous.) A term (or a rewrite rule or a recursion scheme) is *homogeneously typed* just if all types that occur in it are homogeneous.

[3] The functors preserve the decidability of MSO theories. It follows that all structures in the two hierarchies have decidable MSO theories.

(ii) A rewrite rule $F\,y_1 \cdots y_k \to t$ is *safe* just if for each subterm s of t that occurs in the operand position (i.e. as the second argument) of an application, for every $1 \leq j \leq k$, if the parameter y_j occurs in s then $ord(s) \leq ord(y_j)$.
(iii) A recursion scheme is safe just if it is homogeneously typed and all its rewrite rules are safe.

For example, the order-2 rule $F\,\varphi\,x\,y \to f\,(F\,\underline{(F\,\varphi\,y)}\,y\,(\varphi\,x))\,a$, where $F : (o \to o) \to o \to o \to o$ and $f : o \to o \to o$, is unsafe, because the order-0 parameter y occurs in the underlined order-1 subterm, which is in an operand position. Note that it follows from the definition that order-0 and order-1 recursion schemes are always safe. Safety may be regarded as a reformulation of Damm's *derived types* [19]; see de Miranda's thesis [47] for a proof of equivalence.

Remark 16. The definition of safe recursion schemes by Knapik et al. [36] assumes that all types are homogeneous. In an unpublished note, Blum and Broadbent [7] have shown that (i) homogeneity is superfluous for safety, in the sense that for generating ranked trees, homogeneously-typed safe recursion schemes are equi-expressive as safe recursion schemes, (ii) homogeneity is superfluous in general i.e. homogeneously-typed unsafe recursion schemes are equi-expressive as unsafe recursion schemes.

What is the point of safety? Though somewhat unnatural as a syntactic constraint, safety does have a clear algorithmic value. It is a well-known fact of symbolic logic and formal systems such as the lambda calculus that *capture-permitting substitution is unsound*. Therefore, when performing substitution, one must avoid variable capture, for example, by renaming bound variables. Remarkably, in safe recursion schemes, it is a relatively straightforward consequence of the definition that capture-permitting substitution *is* sound; in other words, it is safe *not* to rename bound variables. It follows that no fresh names are needed when computing the value tree of a safe recursion scheme. Knapik et al. [36] used this fact in their proof of the decidability of the MSO theories of the value trees of safe recursion schemes.

In view of their algorithmic advantage (namely, space efficiency), it is pertinent to ask if safe recursion schemes are less expressive. (We consider this important question in Section 4.3.) The safe lambda calculus [8, 6] is a formalisation of safety as a subsystem of the simply-typed lambda calculus. Blum and Ong [8] showed that, using Church numerals, the numeric functions representable by simply-typed safe lambda-terms are exactly the multivariate polynomials. This should be contrasted with the classical result of Schwichtenberg [65]: the numeric functions representable by simply-typed lambda-terms are the multivariate polynomials augmented with conditionals. For a systematic study of the safe lambda calculus, including its expressivity, complexity and semantics, see Blum's doctoral thesis [6].

3 Model Checking Higher-Order Recursion Schemes

What classes of infinite structures have decidable monadic second-order (MSO) theories? One of the best known examples of such a decidable class are the *reg-*

ular trees as studied by Rabin in a landmark paper in 1969 [63]. Muller and Shupp [48] subsequently proved that the *configuration graphs of pushdown systems* also have decidable MSO theories. In the 1990s, as finite-state technologies matured, researchers embraced the challenges of software and hence infinite-state model checking. A highlight in this period was Caucal's result [11] that *prefix-recognisable graphs* (equivalently the ϵ-closure of configuration graphs of pushdown systems) have decidable MSO theories. Another concerned *algebraic trees*, which are trees generated by context-free tree grammars. Courcelle [15] showed that these trees have decidable MSO theories. In 2002, work by Knapik et al. [36] and Caucal [12] significantly extended and unified earlier developments.

Theorem 17 (Knapik et al. 2002 & Caucal 2002). *For each $n \geq 0$, all trees in $\mathit{SafeRecTree}_n^{\Sigma} = \mathit{PushdownTree}_n\Sigma = \mathit{CaucalTree}_n^{\Sigma}$ have decidable MSO theories.*

We give an outline of the proof [36] which is by induction on n. Given an order-$(n+1)$ safe recursion scheme G, consider an associated tree, call it $\beth_G$, which is obtained by contracting all the order-1 β-redexes in the rewrite rules of G. The tree $\beth_G$ coincides with the tree generated by an order-n recursion scheme G^{α} i.e. $\beth_G = [\![G^{\alpha}]\!]$; further the MSO theory of the original order-$(n+1)$ tree $[\![G]\!]$ is reducible to that of the order-n tree $[\![G^{\alpha}]\!]$ i.e. there is a computable transformation of MSO sentences $\varphi \mapsto \varphi'$ such that $[\![G]\!] \models \varphi$ iff $[\![G^{\alpha}]\!] \models \varphi'$ [36, Theorem 3.3]. Thanks to the safety assumption, it is sound to contract β-redexes using *capture-permitting* substitution i.e. without renaming bound variables. It follows that one can construct the tree $\beth_G$ using only the original variables of the recursion scheme G. The same construction on an arbitrary recursion scheme would require an infinite set of fresh variable names.

3.1 Some Key Questions

Assuming safety, recursion schemes have decidable MSO theories, and their expressivity is characterised by higher-order pushdown automata. Yet safety is an awkward condition, both syntactically and semantically. Is safety really essential for these desirable properties? We consider a number of key questions.

Question 1 *MSO Decidability*: Do trees generated by recursion schemes have decidable MSO theories?

Question 2 *Automata-Theoretic Characterisation*: Find a class of automata that characterise the expressive power of recursion schemes. Specifically, can higher-order pushdown automata be so extended that they define the same class of ranked trees as recursion schemes?

Question 3 *Graph Families*: Is there a good definition of graphs generated by recursion schemes? We expect the unravelling of these graphs to coincide with the value trees of recursion schemes. What logical theories of these graphs are decidable?

Question 4 *Expressivity*: Does safety really constrain expressivity? Are there *inherently unsafe* word languages / trees / graphs?

The rest of the section is devoted to Question 1. We consider the remaining questions in Section 4.

3.2 MSO Decidability and a Semantic Proof

A first partial answer to Question 1 was obtained by Aehlig et al. [2]; they showed that all trees up to order 2, whether safe or not, and whether homogeneously typed or not, have decidable MSO theories. Independently, Knapik et al. obtained a somewhat sharper result [37]. Subsequently, in a LICS 2006 paper [53], Ong answered the question fully.

Theorem 18 (Ong 2006). *For each $n \geq 0$ the modal mu-calculus model checking problem for trees generated by order-n recursion schemes is n-*EXPTIME *complete.*

Since MSOL and the modal mu-calculus are equi-expressive over trees, it follows that these trees have decidable MSO theories. In the following we sketch a proof of the theorem.

Thanks to Emerson and Jutla [24], we can reduce the mu-calculus model checking problem to an alternating parity tree automaton (APT) acceptance problem: *Given an order-n recursion scheme G and an APT $\mathcal{B}$, does $\mathcal{B}$ have an accepting run-tree over the value tree $[\![G]\!]$?* The proof has two main ingredients.

I. The first is a *transference principle* from the value tree to an auxiliary *computation tree*, which is regular. Using game semantics [32], we establish a strong correspondence between *paths* in the value tree and *traversals* in the computation tree. This allows us to prove that the APT $\mathcal{B}$ has an accepting run-tree over the value tree if and only if it has an accepting *traversal-tree* over the computation tree.

II. The second ingredient is the simulation of an accepting traversal-tree by a certain set of annotated paths over the computation tree: we construct a *traversal-simulating* APT, $\widehat{\mathcal{B}}$, as a recognising device for this set of paths.

Thus we have

$$
\begin{array}{ll}
 & \text{APT } \mathcal{B} \text{ has an accepting run-tree over the value tree } [\![G]\!] \\
\iff & \{ \textit{ I. Transference Principle: Traversal-Path Correspondence} \} \\
 & \text{APT } \mathcal{B} \text{ has an accepting } \textit{traversal-tree} \text{ over the } \textit{computation tree } \lambda(G) \\
\iff & \{ \textit{ II. Simulation of Traversals by Paths } \} \\
 & \text{Traversal-simulating APT } \widehat{\mathcal{B}} \text{ has an accepting run-tree over } \lambda(G)
\end{array}
$$

which is decidable, because the computation tree $\lambda(G)$ is regular.

We elaborate on the key ideas behind the proof. By construction, the value tree is the *extensional* outcome of a potentially infinite process of rewriting. It would be quite futile to analyse the value tree directly, since it has no useful structure for our purpose. Our approach is to consider what we call the *long transform*, of a recursion scheme G, written $\overline{G}$, which is obtained by expanding the right-hand side of each rewrite rule to its η-long form, inserting explicit application operators, and then currying the rule. The transform allows us to

tease out the two constituent actions of the rewriting process, namely, *unfolding* and β-reduction, and hence analyse them separately. See Example 19(i). Thus, take a recursion scheme G.

- We first build an auxiliary *computation tree* $\lambda(G)$ which is obtained by performing all of the unfolding but none of the β-reduction in the rewrite rules of G. Formally $\lambda(G)$ is defined to be the value tree of the long transform $\overline{G}$, which is an order-0 recursion scheme by construction. See Example 19(ii). As no substitution is performed, no variable-renaming is needed.

- We then analyse the β-reductions *locally* (i.e. without the *global* operation of substitution) using game semantics [32].

Example 19. Fix a ranked alphabet $\Sigma = \{ g : 2, h : 1, a : 0 \}$.

(i) We present the long transform $\overline{G_3}$ of an order-2 (unsafe) recursion scheme G_3:

$$G_3 : \begin{cases} S \to H\, a \\ H\, z \to F\, (g\, z) \\ F\, \varphi \to \varphi\, (\varphi\, (F\, h)) \end{cases} \quad \mapsto \quad \overline{G_3} : \begin{cases} S \to \lambda.@\, H\, (\lambda.a) \\ H \to \lambda z.@\, F\, (\lambda y.g\, (\lambda.z)\, (\lambda.y)) \\ F \to \lambda\varphi.\varphi\, (\lambda.\varphi\, (\lambda.@\, F\, (\lambda x.h\, (\lambda.x)))) \end{cases}$$

The value tree $[\![G_3]\!]$ is the tree on left in Figure 1. The only infinite path in the tree is 221^ω.

(ii) The computation tree $\lambda(G_3)$ of G_3 is the tree on the right in Figure 1. In the figure, for ease of reference, we give nodes of $\lambda(G_3)$ numeric names (in square-brackets).

(iii) With reference to Figure 1, the maximal traversals (pointers omitted) over the computation tree $\lambda(G_3)$ are:

$$\begin{array}{l} 0\ 1\ 2\ 3\ 4\ 5\ 13\ 14\ 15\ 16\ 19\ 20 \\ 0\ 1\ 2\ 3\ 4\ 5\ 13\ 14\ 17\ 18\ 6\ 7\ 13\ 14\ 15\ 16\ 19\ 20 \\ 0\ 1\ 2\ 3\ 4\ 5\ 13\ 14\ 17\ 18\ 6\ 7\ 13\ 14\ 17\ 18\ 8\ 9\ 10\ 21\ 11\ 12 \cdots \end{array}$$

They correspond respectively to the paths $g\ a$, $g\ g\ a$ and $g\ g\ h^\omega$ in $[\![G]\!]$.

Note that we do not (need to) assume that the recursion scheme G is safe. We can now state a strong correspondence between *paths* in the value tree and *traversals* [53, 54] over the computation tree. See Example 19(iii).

Theorem 20 (Path-Traversal Correspondence). *Fix a ranked alphabet Σ. Let G be an order-n recursion scheme.*

(i) There is a 1-1 correspondence between maximal paths p in the value tree $[\![G]\!]$ and maximal traversals t_p over the computation tree $\lambda(G)$.
(ii) Further, for each p, we have $p \restriction \Sigma = t_p \restriction \Sigma$.

The theorem is proved using game semantics [54, 32]. In the language of game semantics, paths in the value tree correspond exactly to P-views in the strategy-denotation of the recursion scheme; a traversal is then (a representation of)

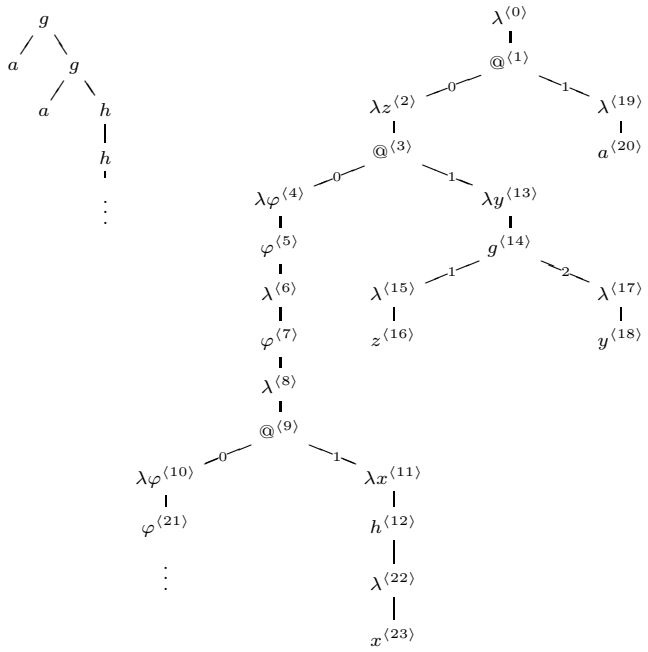


Fig. 1. The value tree $[\![G_3]\!]$ and computation tree $\lambda(G_3)$ of Example 19.

the *uncovering*[4] of such a play. The path-traversal correspondence allows us to prove that a given alternating parity tree automaton (APT) has an accepting run-tree over the value tree if and only if it has an accepting *traversal-tree* over the computation tree.

Our problem is then reduced to finding an effective method of recognising certain sets of infinite traversals over a given computation tree that satisfy the parity condition. This requires a new idea as a traversal is most unlike a path; it can jump all over the tree and may even visit certain nodes infinitely often. Our solution again exploits the game-semantic connexion. It is a property of traversals that their *P-views* are paths (in the computation tree). This allows us to simulate a traversal over a computation tree by the P-views of its prefixes, which are annotated paths of a certain kind in the same tree. The simulation is made precise in the notion of *traversal-simulating* APT. We establish the correctness of the simulation by proving that a given APT $\mathcal{B}$ has an accepting traversal-tree over the computation tree if and only if the associated *traversal-simulating* APT $\widehat{\mathcal{B}}$ has an accepting run-tree over the computation tree. The

[4] In game semantics [32], plays in the composite strategy $\sigma;\tau : A \to C$ are constructed from those in $\sigma : A \to B$ and $\tau : B \to C$ by "parallel composition plus hiding" [31]. By construction, every play s in the composite strategy $\sigma\,;\tau$ is obtained from an *interaction sequence* $\widehat{s}$ — which is a certain sequence of moves-with-pointer from arenas A, B and C — by hiding all moves of B. We call $\widehat{s}$ the *uncovering* of s.

control-states of a traversal-simulating APT are *variables profiles*, which are assertions concerning variables about the APT-state being simulated and the largest priority encountered during a relevant part of the computation.

Decidability of the modal mu-calculus model checking problem for trees generated by recursion schemes follows at once since computation trees are regular, and the APT acceptance problem for regular trees is decidable [63, 24].

Finally, to prove n-EXPTIME decidability of the model checking problem, we first establish a certain *succinctness property* for traversal-simulating APT: If a traversal-simulating APT has an accepting run-tree, then it has one with a reduced branching factor. The desired time bound is then obtained by analysing the complexity of solving an associated (finite) acceptance parity game, which is an appropriate product of the traversal-simulating APT and a finite deterministic graph that unravels to the computation tree in question.

3.3 Other Proofs of the Decidability Theorem

Several other proofs of Theorem 18 have been published.

In a LICS 2008 paper [29], Hague et al. gave a proof by reducing the modal mu-calculus model checking of recursion schemes to the solution of parity games over the configuration graphs of collapsible pushdown automata (see Theorem 32 in the sequel). The proof of correctness of the scheme-to-automaton transformation (Theorem 24) uses game semantics.

In a LICS 2009 paper [40], Kobayashi and Ong gave a proof that uses type theory. They exhibit a transformation that, given an APT $\mathcal{A}$, constructs a type system $\mathcal{K}_{\mathcal{A}}$ such that a recursion scheme is typable in $\mathcal{K}_{\mathcal{A}}$ if and only if the value tree of the recursion scheme is accepted by the APT $\mathcal{A}$. The model checking problem is thus reduced to a type inference problem. An advantage of this model checking algorithm is that it has an improved parameterised complexity: the time complexity is polynomial in the size of the recursion scheme, assuming that the types and the APT are fixed.

Salvati and Walukiewicz [64] recently gave yet another proof of Theorem 18. They consider trees generated by the λY-calculus, and use Krivine machine as a model of computation. Their proof is by reducing the problem of solving a parity game over the configurations of a Krivine machine to that of solving a finite parity game.

4 Collapsible Pushdown Automata

In this section, we consider Questions 2, 3 and 4 of Section 3.1. We introduce a variant of higher-order pushdown automata, called *collapsible pushdown automata*, and show that they generate the same class of trees as recursion schemes. We then define the configuration graphs of collapsible pushdown systems, and state a result on the solution of parity games over these graphs. Finally we discuss recent progress on the Safety Conjecture.

4.1 Collapsible Pushdown Automata

Collapsible pushdown automata (CPDA) are a variant of higher-order pushdown automata in which every symbol in the stack has a link to a prefix of the stack. In addition to the higher-order stack operations $push_i$ and pop_i, CPDA have an important operation called *collapse*, whose effect is to "collapse" the stack s to the prefix indicated by the link from the $top_1(s)$. The main result is that for every $n \geq 0$, order-n recursion schemes and order-n CPDA are equi-expressive as generators of ranked trees.

Let Γ be a stack alphabet and $n \geq 1$. An *order-n collapsible stack* s is an order-n stack such that every non-$\bot$ symbol that occurs in it has a link to a collapsible stack (of order k) situated below it in s; we call the link a $(k+1)$-*link*. Note that k is necessarily less than n. We shall abbreviate order-n collapsible stack to n-stack, whenever it is clear form the context. The *empty k-stack* is defined as before. When displaying examples of n-stacks, we shall omit $\bot$ and 1-links (i.e. links to stack symbols) to avoid clutter; thus we write $[[]\,[a\,b]]$ instead of $[[\bot\ a\ b]]$

For $n \geq 2$ the set $Op_n^\dagger$ of *order-n operations on collapsible stacks* consists of the following four types of operations:

(i) pop_k for each $1 \leq k \leq n$
(ii) *collapse*
(iii) $push_1^{a,k}$ for each $1 \leq k \leq n$ and each $a \in (\Gamma \setminus \{\bot\})$
(iv) $push_j$ for each $2 \leq j \leq n$.

The operation pop_i is defined as before in Section 2.1. Let s be an n-stack and $2 \leq i \leq n$. To construct $push_1^{a,i}\, s$, we first attach a link from a fresh copy of a to the $(i-1)$-stack that is immediately below the top $(i-1)$-stack of s, and then push the symbol-with-link onto the top 1-stack of s. As for *collapse*, suppose the top_1-symbol of s has a link to a (particular occurrence of) k-stack u in s. Then *collapse* s causes s to "collapse" to the prefix s_0 of s such that $top_{k+1}\, s_0 = u$. Finally, for $j \geq 2$, the *order-j push* operation, $push_j$, simply takes a stack s and duplicates the top $(j-1)$-stack of s, preserving its link structure.

Example 21. Take the 3-stack $s = [[[a]]\ [[]\,[a]]]$. We have

$$push_1^{b,2}\, s = [[[a]]\ [[]\,[a\ b]]] \qquad collapse\,(push_1^{b,2}\, s) = [[[a]]\ [[]]]$$

$$\underbrace{push_1^{c,3}(push_1^{b,2}\, s)}_{\theta} = [[[a]]\ [[]\,[a\ b\ c]]]$$

Then $push_2\,\theta$ and $push_3\theta$ are respectively

$$[[[a]]\ [[]\,[a\ b\ c]\,[a\ b\ c]]] \quad \text{and} \quad [[[a]]\ [[]\,[a\ b\ c]]\ [[]\,[a\ b\ c]]].$$

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

As in the case of order-n stacks, the tuple $\langle \Gamma, n\text{-}Stack^{\dagger}_{\Gamma}, Op^{\dagger}_{n}, top_1, \bot_n \rangle$ is called the *system of order-n collapsible stacks*, which is an abstract store system. We shall use automata equipped with order-n collapsible stacks to define word languages and trees, and (in Section 4.2) infinite graphs.

Definition 22. Let $\mathcal{S} = \langle \Gamma, n\text{-}Stack^{\dagger}_{\Gamma}, Op^{\dagger}_{n}, top_1, \bot_n \rangle$ be the system of order-n collapsible stacks over Γ. An *order-n collapsible pushdown word-language automaton* is just a word-language $\mathcal{S}$-automaton $\langle \mathcal{S}, Q, \Sigma, \Delta, q_I, F \rangle$, and we specify it as $\langle \Gamma, Q, \Sigma, \Delta, q_I, F \rangle$. Similarly an *order-n collapsible pushdown tree automaton* is just a tree $\mathcal{S}$-automaton $\langle \mathcal{S}, Q, \Sigma, \delta, q_I \rangle$, and we specify it as $\langle \Gamma, Q, \Sigma, \delta, q_I \rangle$.

Example 23 (Urzyczyn 2003; Aehlig, de Miranda and Ong 2005).

(i) We define the language U over the alphabet $\{ \mathbf{(}, \mathbf{)}, * \}$ as follows. A *U-word* is composed of 3 segments:

$$\underbrace{\mathbf{(}\cdots\mathbf{(}\cdots\mathbf{(}}_{A}\ \underbrace{\mathbf{(}\cdots\mathbf{)}\cdots\mathbf{(}\cdots\mathbf{)}}_{B}\ \underbrace{*\cdots*}_{C}$$

- Segment A is a prefix of a well-bracketed word that ends in $\mathbf{(}$, and the opening $\mathbf{(}$ is not matched in the (whole) word.
- Segment B is a well-bracketed word.
- Segment C has length equal to the number of $\mathbf{(}$ in A.

It is a consequence of the definition that every U-word has a unique decomposition. For example, $\mathbf{(}\,\mathbf{)}\,\mathbf{(}\,\underline{\mathbf{(}\,\mathbf{)}\,\mathbf{(}\,\mathbf{)}}\,\mathbf{)} * * *$ is in U; its B-segment is underlined. For each $n \geq 0$, the word $\mathbf{(}\,\overline{\mathbf{(}^{n}\,\mathbf{)}^{n}\,\mathbf{(}}\ *^{n}\ **$ is in U, the respective B-segments are all empty.

(ii) The language U is recognisable by a *deterministic* order-2 collapsible pushdown automaton $\langle \{ q_I, q_1, q_2 \}, \{ \mathbf{(}, \mathbf{)}, * \}, \{ \bot, Z \}, \delta, q_I, \{ q_2 \} \rangle$, where $\delta : Q \times \Sigma \times \Gamma \to Op^{*}_{n} \times Q$ is as follows:

$$\begin{array}{ll} (q_I, \mathbf{(}, \bot) \mapsto (push_2 \,;\, push^{Z}_{1}, q_1) & (q_1, *, Z) \mapsto (collapse, q_2) \\ (q_1, \mathbf{(}, Z) \mapsto (push_2 \,;\, push^{Z}_{1}, q_1) & (q_2, *, Z) \mapsto (pop_2, q_2) \\ (q_1, \mathbf{)}, Z) \mapsto (pop_1, q_1) & \end{array}$$

The idea is that the pair $(q_1, Z) \in Q \times \Gamma$ indicates that the number of "$\mathbf{(}$" read, minus the number of "$\mathbf{)}$" read, is at least one. Note that $(q_1, \bot)$ indicates a "stuck configuration" which is reachable upon reading e.g. $\mathbf{(}\,\mathbf{)}$. To illustrate, we present the computation of the U-word $\mathbf{(}\,\mathbf{(}\,\mathbf{)}\,\mathbf{(}\,\mathbf{(}\,\mathbf{)}\ ***$ in Figure 2. (In the figure, we omit the unimportant links.)

It follows from [3] that U is recognisable by a *non-deterministic* order-2 pushdown automaton. This illustrates the power of collapse.

The main result of this section is the following equi-expressivity result.

Theorem 24 (Hague, Murawski, Ong and Serre 2008). *For every $n \geq 0$, order-n recursion schemes and order-n collapsible pushdown tree automata define the same class of Σ-labelled trees.*

The proof is in the long version of [29]. Here we explain the main ideas.

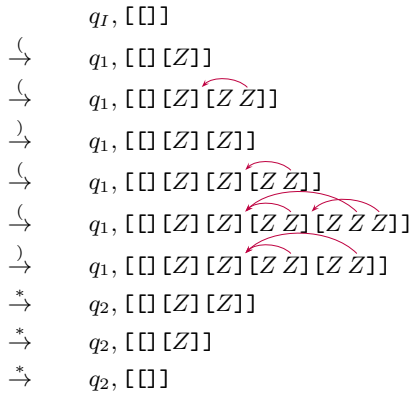


Fig. 2. The computation of the U-word **(( ) (( )** $* * *$.

From CPDA to Recursion Schemes We construct an algorithm that transforms a given order-n CPDA to an equivalent order-n recursion scheme. To illustrate, we present the algorithm in the order-2 case, which is due to Knapik et al. [37]. For a generalisation to all finite orders, see [29]. Fix an order-2 CPDA $\mathcal{A}$ with state-set $\{0, \cdots, m-1\}$. Let 0 be the base type; we define $n+1 := n^m \to n$ where $A^m \to B$ is a shorthand for the type $A \to \cdots \to A \to B$ (m occurrences of A). For each stack symbol Z and each state $0 \leq p \leq m-1$, we introduce a non-terminal

$$\mathcal{F}_p^Z \ : \ 0^m \to 1^m \to 0^m \to 0$$

that represents the (top-of-stack) symbol Z with a (order-2) link in state p. In addition, for $i \in \{0, 1\}$, we introduce a non-terminal $\Omega_i : i$, and fix a start symbol $S : 0$. Let $P(i)$ be a term with an occurrence of (the parameter) i; we write $\langle P(i) \mid i \rangle$ as a shorthand for the sequence $P(0) \cdots P(m-1)$. For example $\langle \mathcal{F}_i^Z \mid i \rangle$ denotes the sequence $\mathcal{F}_0^Z \cdots \mathcal{F}_{m-1}^Z$.

We briefly explain the idea of the translation. The intuition is that a term of the form $\mathcal{F}_p^Z \, \overline{L} \, \overline{M_1} \, \overline{M_0} \ : \ 0$ represents a configuration (p, s) where p is a state and s is a 2-stack; equivalently the sequence $\langle \mathcal{F}_i^Z \, \overline{L} \, \overline{M_1} \, \overline{M_0} \mid i \rangle$ represents the 2-stack s (the state is "abstracted" by the sequence). Further

- $(p, top_1\, s)$ — where the top_1-symbol of s is Z which has a link to the 1-stack represented by $\overline{L} : 0^m$ — is represented by $\mathcal{F}_p^Z \, \overline{L} : 2$
- $(p, top_2\, s)$ is represented by $\mathcal{F}_p^Z \, \overline{L} \, \overline{M_1} : 1$
- $(p, pop_2\, s)$ is represented by $M_{0,p} : 0$ and $(p, pop_1\, s)$ by $M_{1,p} \, \overline{M_0} : 0$
- $(p, collapse\ s)$ is represented by $L_p : 0$.

Definition 25. The order-2 *recursion scheme determined by* $\mathcal{A}$, written $G_{\mathcal{A}}$, has the following rewrite rules. We use vector notation; for example $\overline{\psi_1}$ is a shorthand for the sequence $\psi_{1,0} \cdots \psi_{1,m-1}$.

- Start rule: $S \to \mathcal{F}_0^{\perp} \, \overline{\Omega_0} \, \overline{\Omega_1} \, \overline{\Omega_0}$

- For each $(p, Z, q, \theta) \in \delta$: $\mathcal{F}_p^Z\ \overline{\varphi}\ \overline{\psi_1}\ \overline{\psi_0} \ \rightarrow\ \Xi_{(q,\theta)}$ where $\Xi_{(q,\theta)}$ is defined by

(q,θ)	$\Xi_{(q,\theta)}$
$(q, push_1^Y)$	$\mathcal{F}_q^Y\ \overline{\psi_0}\ \langle \mathcal{F}_i^Z\ \overline{\varphi}\ \overline{\psi_1} \mid i\rangle\ \overline{\psi_0}$
$(q, push_1)$	$\mathcal{F}_q^Z\ \overline{\varphi}\ \langle \mathcal{F}_i^Z\ \overline{\varphi}\ \overline{\psi_1} \mid i\rangle\ \overline{\psi_0}$
$(q, push_2)$	$\mathcal{F}_q^Z\ \overline{\varphi}\ \overline{\psi_1}\ \langle \mathcal{F}_i^Z\ \overline{\varphi}\ \overline{\psi_1}\ \overline{\psi_0} \mid i\rangle$

(q,θ)	$\Xi_{(q,\theta)}$
(q, pop_1)	$\psi_{1,q}\ \overline{\psi_0}$
(q, pop_2)	$\psi_{0,q}$
$(q, collapse)$	φ_q

- For each $(p, Z, f, \overline{q}) \in \delta$: $\mathcal{F}_p^Z\ \overline{\varphi}\ \overline{\psi_1}\ \overline{\psi_0} \ \rightarrow\ f\ (\mathcal{F}_{q_1}^Z\ \overline{\varphi}\ \overline{\psi_1}\ \overline{\psi_0})\ \cdots\ (\mathcal{F}_{q_{\mathsf{ar}(f)}}^Z\ \overline{\varphi}\ \overline{\psi_1}\ \overline{\psi_0})$.

Example 26. Consider the language U and the order-2 CPDA defined in Example 23. Applying the transformation, we obtain an order-2, deterministic, unsafe recursion scheme over the ranked alphabet $\{\,\mathbf{(}:1, \mathbf{)}:1, *:1, e:0\,\}$ that generates U.

$$
\begin{array}{rcl}
S & \rightarrow & \mathcal{F}_0^{\perp}\ \overline{\Omega_0}\ \overline{\Omega_1}\ \overline{\Omega_0} \\
\mathcal{F}_0^{\perp}\ \overline{z}\ \overline{\varphi}\ \overline{x} & \rightarrow & \mathbf{(}\ \ \big(\mathcal{F}_1^Z\ \langle F_i^{\perp}\ \overline{z}\ \overline{\varphi}\ \overline{x} \mid i\rangle\ \langle F_i^{\perp}\ \overline{z}\ \overline{\varphi} \mid i\rangle\ \langle F_i^{\perp}\ \overline{z}\ \overline{\varphi}\ \overline{x} \mid i\rangle\big) \\
\mathcal{F}_1^Z\ \overline{z}\ \overline{\varphi}\ \overline{x} & \rightarrow & \mathbf{(}\ \ \big(\mathcal{F}_1^Z\ \langle F_i^Z\ \overline{z}\ \overline{\varphi}\ \overline{x} \mid i\rangle\ \langle F_i^Z\ \overline{z}\ \overline{\varphi} \mid i\rangle\ \langle F_i^Z\ \overline{z}\ \overline{\varphi}\ \overline{x} \mid i\rangle\big) \\
\mathcal{F}_1^Z\ \overline{z}\ \overline{\varphi}\ \overline{x} & \rightarrow & \mathbf{)}\ (\varphi_1\ \overline{x}) \\
\mathcal{F}_1^Z\ \overline{z}\ \overline{\varphi}\ \overline{x} & \rightarrow & *\ z_2 \\
\mathcal{F}_2^Z\ \overline{z}\ \overline{\varphi}\ \overline{x} & \rightarrow & *\ x_2 \\
\mathcal{F}_2^{\perp}\ \overline{z}\ \overline{\varphi}\ \overline{x} & \rightarrow & e
\end{array}
$$

From Recursion Schemes to CPDA Our proof uses the theory of traversals [54, 6], which is based on game semantics [32].

Let G be an order-n recursion scheme. From the computation tree $\overline{G}$, we define a labelled directed graph $\mathrm{Gr}(G)$, which will serve as a blueprint for the definition of $\mathsf{CPDA}(G)$, the CPDA determined by G. To construct $\mathrm{Gr}(G)$, we first take the forest consisting of the abstract syntactic tree of the right-hand sides of $\overline{G}$. We orient the edges towards the leaves. For each node with label f (say), the outgoing edges are labelled with directions $1, 2, \cdots, \mathsf{ar}(f)$ respectively, except that edges from nodes labelled by @ are labelled from 0. Let us write $v = E_i(u)$ just if (u, v) is an edge labelled by i. Next, for every non-terminal F, we identify the root rt_F of the abstract syntactic tree of the right-hand side of the rule for F with all nodes labelled F (which were leaves in the forest). We designate the node rt_S, where S is the start symbol of $\overline{G}$, as the root of $\mathrm{Gr}(G)$. The graph $\mathrm{Gr}(G_3)$ for the order-2 recursion scheme G_3 of Example 19 is shown in Figure 3.

We are now ready to describe $\mathsf{CPDA}(G)$. The set of nodes of $\mathrm{Gr}(G)$ will become the stack alphabet of $\mathsf{CPDA}(G)$. The initial configuration will be the n-stack $push_1^{v_0,1}\ \perp_n$, where v_0 is the root of $\mathrm{Gr}(G)$. For ease of explanation, we define the transition map δ as a function that takes a node $u \in \mathrm{Gr}(G)$ to a sequence of stack operations, by a case analysis of the label l_u of u. When l_u is not a variable, the action is just $push_1^{v,1}$, where v is an appropriate successor of the node u. Precisely, we define v as follows.

$$
v\ :=\ \begin{cases} E_0(u) \text{ if } l_u = @ \\ E_1(u) \text{ if } l_u = \lambda\overline{\varphi} \\ E_i(u) \text{ if } l_u \in \Sigma \end{cases}
$$

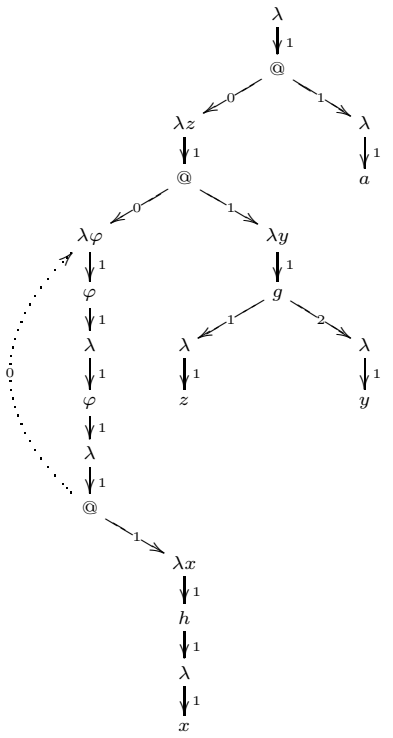


Fig. 3. The graph $\mathrm{Gr}(G_3)$ determined by the order-2 recursion scheme G_3.

where i is the direction that the automaton is to explore in the value tree.

Finally, suppose l_u is a variable φ_i and its binder is a lambda node $\lambda\overline{\varphi}$ which is in turn a j-child.

- If φ has order $l \geq 1$ we define

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

where $push_1^{E_i(top_1),k}$ is defined to be the operation $s \mapsto push_1^{E_i(top_1\, s),k}\, s$.

- If φ has order 0 we define

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

It can be shown that the runs of $\mathsf{CPDA}(G)$ are in 1-1 correspondence with traversals, in the sense of Ong [53] (see the systematic treatment [54]). Since

traversals are *uncoverings* [32] of paths in the value tree $[\![G]\!]$, for every order-n recursion scheme G, the order-n CPDA-transform, $\mathsf{CPDA}(G)$, generates the value tree $[\![G]\!]$.

Remark 27. (i) Since the construction of the CPDA-transform in the scheme-to-automata translation is based on game semantics [32], Theorem 24 also gives an automata-theoretic characterisation of innocent strategies. There are several machine-theoretic representations of innocent game semantics in the literature: PAM, the Pointer Abstract Machine of Danos et al. [22], may be viewed as an implementation of linear head reduction of lambda-terms; Curien and Herbelin [16, 17] used abstract Böhm trees as the basis of a family of abstract machines. Compared to these machines, the characterisation by CPDA seems clearly syntax-independent: contrast, for example, the type-theoretic notion of order with the order of collapsible stacks. (ii) Blum and Broadbent [7] recently proved that if the recursion scheme G is safe, then the CPDA-transform, $\mathsf{CPDA}(G)$, does not use *collapse* in its computation.

In a LICS 2012 paper [10], Carayol and Serre gave a syntactic proof of Theorem 24. Their scheme-to-CPDA translation does not use game semantics. They show that if the recursion scheme is safe, then the CPDA-translate does not use *collapse* in its computation.

4.2 Parity Games over CPDA Configuration Graphs

Definition 28. (i) Let $\mathcal{S} = \langle \Gamma, \mathit{AStore}_\Gamma, \mathit{Op}, \mathit{top}, \bot \rangle$ be an abstract store system. An *$\mathcal{S}$-transition system* is a tuple $\mathcal{T} = \langle \mathcal{S}, Q, \Delta, q_I \rangle$ where Q is a finite set of control-states, $q_I \in Q$ is the initial state, and $\Delta \subseteq Q \times \Gamma \times Q \times \mathit{Op}$ is the transition relation. A *configuration* is a pair (q, s) where $q \in Q$ and $s \in \mathit{AStore}_\Gamma$; and $(q_I, \bot)$ is the initial configuration. The transition relation Δ induces a (labelled) transition relation between configurations according to the rule: $(q, s) \xrightarrow{(q', \theta)} (q', \theta(s))$ provided $(q, \mathit{top}(s), q', \theta) \in \Delta$. The *configuration graph* of $\mathcal{T}$ is a directed graph whose vertices are the configurations, and edge-set is the induced transition relation.

(ii) In case $\mathcal{S} = \langle \Gamma, n\text{-}\mathit{Stack}_\Gamma, \mathit{Op}_n, \mathit{top}_1, \bot_n \rangle$ is the system of order-n stacks over Γ, we refer to a $\mathcal{S}$-transition system $\mathcal{T} = \langle \mathcal{S}, Q, \Delta, q_I \rangle$ as an *order-n pushdown system* (order-n PDS).

(iii) Similarly, in case $\mathcal{S} = \langle \Gamma, n\text{-}\mathit{Stack}^\dagger_\Gamma, \mathit{Op}^\dagger_n, \mathit{top}_1, \bot_n \rangle$ is the system of order-n collapsible stacks over Γ, we call a $\mathcal{S}$-transition system $\mathcal{T} = \langle \mathcal{S}, Q, \Delta, q_I \rangle$ as an *order-n collapsible pushdown system* (order-n CPDS).

Example 29 (An undecidable CPDS graph). Take the order-2 CPDS with state-set $\{ 0, 1, 2 \}$, stack alphabet $\{ a, b, \bot \}$ and transition relation given by

$$\{ (0, -, 1, t),\ (1, -, 0, a),\ (1, -, 2, b),\ (2, \dagger, 2, 1),\ (2, \dagger, 0, 0) \}$$

where $-$ means any symbol, $\dagger$ means any non-$\bot$ symbol, and $t, a, b, 0$ and 1 are shorthand for the stack operations push_2, $\mathit{push}_1^{a,2}$, $\mathit{push}_1^{b,2}$, $\mathit{collapse}$ and pop_1

respectively. We present its configuration graph (with edges labelled by stack operations only) in Figure 4.

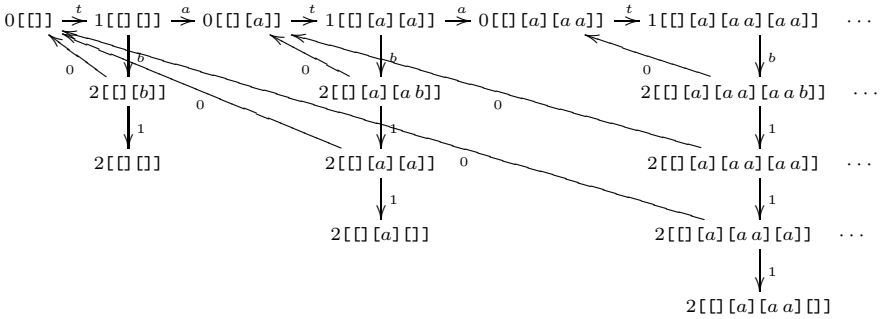


Fig. 4. A order-2 CPDS graph with an undecidable MSO theory

Let $G = \langle V, E \rangle$ be the configuration graph of an $\mathcal{S}$-transition system $\mathcal{A}$, and $Q_{\mathbf{E}} \cup Q_{\mathbf{A}}$ be a partition of Q, and let $\Omega : Q \to \{ 0, \cdots M-1 \}$ be a priority function. Together they define a partition $V_{\mathbf{E}} \cup V_{\mathbf{A}}$ of V whereby a vertex belongs to $V_{\mathbf{E}}$ if and only if its control state belongs to $Q_{\mathbf{E}}$, and a priority function $\Omega : V \to \{ 0, \cdots, M-1 \}$ where a vertex is assigned the priority of its control state. We call the structure $\mathcal{G} = \langle G, V_{\mathbf{E}}, V_{\mathbf{A}} \rangle$ an *order-n CPDS game graph* and the pair $\mathbb{G} = \langle \mathcal{G}, \Omega \rangle$ an *order-n CPDS parity game.*

In this section we consider the problem:

($\mathbf{P}_1$) *Given an order-n CPDS parity game decide if Éloïse has a winning strategy from the initial configuration.*

The Problem ($\mathbf{P}_1$) is closely related to the following problems:

($\mathbf{P}_2$) *Given an order-n CPDS graph G, and a modal mu-calculus formula φ, does φ hold at the initial configuration of G?*

($\mathbf{P}_3$) *Given an APT and an order-n CPDS graph G, does the APT accept the unravelling of G?*

($\mathbf{P}_4$) *Given an MSO formula φ and an order-n CPDS graph G, does φ hold at the root of the unravelling of G?*

Using the techniques of Emerson and Jutla [24], it is straightforward to show that Problem ($\mathbf{P}_1$) is polynomially equivalent to Problems ($\mathbf{P}_2$) and ($\mathbf{P}_3$); and Problem ($\mathbf{P}_1$) is equivalent to Problem ($\mathbf{P}_4$) – the reduction from ($\mathbf{P}_1$) to ($\mathbf{P}_4$) is polynomial, but non-elementary in the other direction.

A useful fact is that the unravelling of an order-n CPDS graph is actually generated by an order-n collapsible pushdown tree automaton (putting labels on the edges makes the order-n CPDS graph *deterministic* and hence its unravelling as desired). Thus an important consequence of Theorem 24 is the following.

Proposition 30. *Let t be the value tree of an order-n recursion scheme. Consider the following problems:*
($\mathbf{P}'_2$) *Given t and a modal mu-calculus formula φ, does φ hold at the root of t?*
($\mathbf{P}'_3$) *Given t and an APT, does the automaton accept t?*
($\mathbf{P}'_4$) *Given t and an MSO formula φ, does φ hold at the root of t?*
Then problem ($\mathbf{P}_i$) is polynomially equivalent to problem ($\mathbf{P}'_i$) for $i = 2, 3$ and 4.

Since the modal mu-calculus model checking problem for value trees of recursion schemes is decidable [53], we obtain the following as an immediate consequence.

Theorem 31. *($\mathbf{P}_1$), ($\mathbf{P}_2$), ($\mathbf{P}_3$) and ($\mathbf{P}_4$) are n-EXPTIME complete.*

Another consequence of Theorem 24 is that it gives new techniques for model checking or solving games played on infinite structures generated by automata. In particular it leads to new proofs/optimal algorithms for the special cases that have been considered previously [67, 66, 37]. Conversely, as Theorem 24 works in both directions, we note that a solution of Problem ($\mathbf{P}_1$) would give a new proof of the decidability of Problems ($\mathbf{P}'_2$), ($\mathbf{P}'_3$) and ($\mathbf{P}'_4$), and would give a new approach to problems on recursion schemes. Actually, the techniques of Walukiewicz [67] and Knapik et al. [37] can be generalised to solve order-n CPDS parity games without reference to Ong's work [53]. Further they give effective winning strategies for the winning player (which was not the case in [37] where the special case $n = 2$ was considered).

Theorem 32 (Hague, Murawski, Ong and Serre 2008). *The problem of solving an order-n CPDS parity game is n-EXPTIME complete. Further one can build an order-n collapsible pushdown transducer (i.e. automaton with output) that realises a winning strategy for the winning player.*

Remark 33. This result can be generalised to the case where the game has an arbitrary ω-regular winning condition, and is played on the ϵ-closure of the configuration graph of an order-n CPDS graph. Consequently parity games on Caucal graphs [12, 66] are a special case of this problem.

The Caucal graphs have decidable MSO theories [12]. Do the configuration graphs of CPDS also have decidable MSO theories?

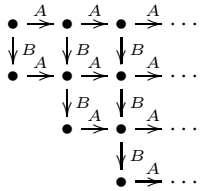
Theorem 34 (Hague, Murawski, Ong and Serre 2008). *There is an order-2 CPDS whose configuration graph has an undecidable MSO theory. Hence the class of ϵ-closure of configuration graphs of CPDS strictly contains the Caucal graphs.*

For a proof, recall that MSO interpretation preserves MSO decidability. Now consider the following MSO interpretation I of the configuration graph of the order-2 CPDS in Example 29:

$$I_A(x, y) = x \xrightarrow{C} y \wedge x \xrightarrow{R} y \qquad I_B(x, y) = x \xrightarrow{1} y$$

with $C = \overline{1}^* \, \overline{b} \, a \, t \, b \, 1^*$ and $R = 0 \, t \, a \, \overline{0} \vee \overline{1} \, 0 \, t \, a \, \overline{0} \, 1$.

Note that for the A-edges, the constraint C requires that the target vertex should be in the next column to the right, while R specifies the correct row. Observe that I's image is the "infinite half-grid" which has an undecidable MSO theory.



Winning Regions and Logical Reflection Broadbent et al. [9] gave the first characterisation of the winning regions of order-n CPDS parity games: they are regular sets defined by a new class of automata. As a corollary, it is shown that recursion schemes are *reflective* with respect to MSOL and modal mu-calculus i.e. there is an algorithm that transforms a given property φ and a recursion scheme G to a new recursion scheme G^φ that *reflects* the property in that $[\![G^\varphi]\!]$ has the same underlying tree as $[\![G]\!]$ except that the nodes that satisfy φ have a special label. Thus we may view G^φ as a transform of G that can internally observe its behaviour against a specification φ.

4.3 Expressivity and the Safety Conjecture

The Safety Conjecture [36] is about the expressivity of *safe* recursion schemes. The conjecture states that there is an *inherently unsafe tree* i.e. there is a tree which is generated by an unsafe recursion scheme, but not by any safe recursion scheme. In view of Theorems 17 and 24, the Safety Conjecture can be stated equivalently in terms of CPDA. As recursion schemes (and CPDA) can be used to define word languages, trees and graphs, it is meaningful to consider the expressivity of safe recursion schemes in each of the three cases.

Aehlig et al. [3] showed that there are no inherently unsafe order-2 word languages: for every unsafe order-2 recursion scheme (respectively order-2 CPDA), there is a safe non-deterministic order-2 recursion scheme (respectively order-2 PDA) that defines the same language. However, Parys [58] has recently shown that the conjecture holds for word languages when restricted to deterministic devices.

Theorem 35 (Parys 2011). *There is a language (similar to U of Example 23) which is recognised by a deterministic order-2 CPDA but not by any deterministic order-n PDA, for any $n \geq 0$.*

The Safety Conjecture (for trees) was recently proved by Parys [59].

Theorem 36 (Parys 2012). *There is a tree which is generated by an order-2 unsafe recursion scheme but not by any safe order-n recursion scheme, for any $n \geq 0$.*

It follows from Theorem 34 that the Safety Conjecture is false in the case of graphs.

5 Application to Model Checking Higher-Order Functional Programs

In a POPL 2009 paper [39], Kobayashi proposed a type-based model checking algorithm for recursion schemes. He considered properties expressible by *trivial automata* [1], which are Büchi tree automata in which every state is final (thus the acceptance condition is trivial). The key result is that given a trivial automaton A, there is an intersection type system $\mathcal{T}_A$ such that for every recursion scheme G, the automaton A accepts the value tree $[\![G]\!]$ if and only if G is typable in $\mathcal{T}_A$. Thus model checking is reduced to type inference. (This type-based approach was subsequently extended by Kobayashi and Ong [40] to a model checking algorithm for all alternating parity tree automata.) Even though trivial automata correspond to a tiny fragment of the modal mu-calculus, the model checking problem is hugely expensive: the complexity remains n-EXPTIME hard [42]. Kobayashi showed that many verification problems of higher-order functional programs, such as reachability, flow analysis and resource usage verification, can be expressed as trivial automata model checking problems of recursion schemes. Thus, the model checking algorithm can serve as a basis for the verification of higher-order functional programs. In a subsequent paper [38], Kobayashi presented a "practical" type inference algorithm. He showed that a tool implementation of the algorithm, called TRecS, performs remarkably well on a range of small but tricky examples, despite the hyper-exponential worst-case complexity.

There has been much progress in higher-order model checking in recent years. Kobayashi [41] and Neatherway, Ramsay and Ong [50] have introduced algorithms for model checking recursion schemes against trivial automata which are inspired by or based on game semantics. There have also been advances in the automatic safety verification of realistic classes of functional programs. Ong and Ramsay [55] have proposed an extension of recursion schemes, called *pattern matching recursion schemes*, which model algebraic data types and function definition by pattern matching, features that are ubiquitous in functional programs. Using counterexample-guided abstraction refinement (CEGAR), they have proposed a sound and semi-complete method for verifying pattern-matching recursion schemes. In a different direction, Kobayashi et al. [43] have formalised predicate abstraction and CEGAR for higher-order model checking, which enable the automatic verification of programs that use infinite data domains such as integers.

Conclusions

Higher-order model checking is challenging and worthwhile. Recursion schemes and collapsible pushdown automata are robust and highly expressive higher-order formalisms for constructing infinite structures. They have rich algorithmic properties. Recent progress in the theory have used semantic methods (such as game semantics and types) as well as automata-theoretic techniques from algorithmic verification. Despite prohibitive worst-case complexity, there are "prac-

tical" model checking algorithms which perform remarkably well on small but tricky examples.

Acknowledgements Much of my work presented here is based on collaboration. I am grateful to all my collaborators for their help. Thanks are also due to EPSRC for financial support.

References

1. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. Logical Methods in Computer Science 3, 1–23 (2007)
2. Aehlig, K., de Miranda, J.G., Ong, C.H.L.: The monadic second order theory of trees given by arbitrary level two recursion schemes is decidable. In: TLCA. pp. 39–54 (2005), lNCS Vol. 3461
3. Aehlig, K., de Miranda, J.G., Ong, C.H.L.: Safety is not a restriction at level 2 for string languages. In: FoSSaCS. pp. 490–501 (2005), lNCS Vol. 3411
4. Aho, A.: Indexed grammars - an extension of context-free grammars. J. ACM 15, 647–671 (1968)
5. de Bakker, J.W., de Roever, W.P.: A calculus for recursive program schemes. In: ICALP. pp. 167–196 (1972)
6. Blum, W.: The Safe Lambda Calculus. Ph.D. thesis, University of Oxford (2008)
7. Blum, W., Broadbent, C.: The CPDA-transform of a safe recursion scheme does not collapse (2009), in preparation
8. Blum, W., Ong, C.H.L.: The safe lambda calculus. LMCS 5(1) (2009)
9. Broadbent, C.H., Carayol, A., Ong, C.H.L., Serre, O.: Recursion schemes and logical reflection. In: LICS. pp. 120–129 (2010)
10. Carayol, A., Serre, O.: Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In: LICS. pp. 165–174 (2012)
11. Caucal, D.: On infinite transition graphs having a decidable monadic theory. In: ICALP. pp. 194–205 (1996), lNCS Vol. 1099
12. Caucal, D.: On infinite terms having a decidable monadic theory. In: MFCS. pp. 165–176 (2002)
13. Courcelle, B.: Fundamental properties of infinite trees. TCS 25, 95–169 (1983)
14. Courcelle, B.: Recursive applicative program schemes. In: Handbook of Theoretical Computer Science, Volume B, pp. 459–492. MIT Press (1990)
15. Courcelle, B.: The monadic second-order logic of graphs IX: machines and their behaviours. Theoretical Computer Science 151, 125–162 (1995)
16. Curien, P.L., Herbelin, H.: Computing with abstract Böhm trees. In: Fuji International Symposium on Functional and Logic Programming. pp. 20–39. World Scientific (1998)
17. Curien, P.L., Herbelin, H.: Abstract machines for dialogue games (2007), coRR abs/0706.2544
18. Damm, W.: Higher type program schemes and their tree languages. Theoretical Computer Science pp. 51–72 (1977)
19. Damm, W.: The IO- and OI-hierarchy. TCS 20, 95–207 (1982)
20. Damm, W., Fehr, E., Indermark, K.: Higher type recursion and self-application as control structures. In: Neuhold, E. (ed.) Formal Descriptions of Programming Concepts, pp. 461–187. North-Holland, Amsterdam (1978)

21. Damm, W., Goerdt, A.: An automata-theoretical characterization of the OI-hierarchy. Information and Control 71, 1–32 (1986)
22. Danos, V., Herbelin, H., Regnier, L.: Game semantics and abstract machines. In: LICS. pp. 394–405 (1996)
23. Duske, J., Parchmann, R.: Linear indexed languages. TCS 32, 47–60 (1984)
24. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: FOCS. pp. 368–377 (1991)
25. Engelfriet, J.: Interated stack automata and complexity classes. Information and Computation 95, 21–75 (1991)
26. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV. pp. 324–336 (2001)
27. Gilman, R.H.: A shrinking lemma for indexed languages. TCS 163, 277–281 (1996)
28. Guessarian, I.: Algebraic Semantics. Springer (1981)
29. Hague, M., Murawski, A.S., Ong, C.H.L., Serre, O.: Collapsible pushdown automata and recursion schemes. In: LICS. pp. 452–461 (2008)
30. Hayashi, T.: On derivation trees of indexed grammars: An extension of the uvwxy-theorem. Publ. RIMS Kyoto Univ. 9, 61–92 (1983)
31. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
32. Hyland, J.M.E., Ong, C.H.L.: On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model. Information and Computation 163, 285–408 (2000)
33. Inaba, K., Maneth, S.: The complexity of tree transducer output languages. In: FSTTCS. pp. 244–255 (2008)
34. Jones, N.D., Muchnick, S.S.: Complexity of finite memory programs with recursion. Journal of the Association for Computing Machinery 25, 312–321 (1978)
35. Kartzow, A., Parys, P.: Strictness of the collapsible pushdown hierarchy. In: MFCS. pp. 566–577 (2012)
36. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: FOSSACS'02. pp. 205–222. Springer (2002), lNCS Vol. 2303
37. Knapik, T., Niwiński, D., Urzyczyn, P., Walukiewicz, I.: Unsafe grammars and panic automata. In: ICALP. pp. 1450–1461 (2005)
38. Kobayashi, N.: Model-checking higher-order programs. In: PPDP. pp. 25–36 (2009)
39. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: POPL. pp. 416–428 (2009)
40. Kobayashi, N., Ong, C.H.L.: A type theory equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: LICS. pp. 179–188 (2009)
41. Kobayashi, N.: A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In: FOSSACS. pp. 260–274 (2011)
42. Kobayashi, N., Ong, C.H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. LMCS 7(4) (2011)
43. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and cegar for higher-order model checking. In: PLDI. pp. 222–233 (2011)
44. Lautemann, C., Schwentick, T., Thérien, D.: Logics for context-free languages. In: Proc. CSL 1994. pp. 205–216. LNCS, Springer-Verlag (1994)
45. Maslov, A.N.: The hierarchy of indexed languages of an arbitrary level. Soviet Math. Dokl. 15, 1170–1174 (1974)
46. Maslov, A.N.: Multilevel stack automata. Problems of Information Transmission 12, 38–43 (1976)
47. de Miranda, J.: Structures generated by higher-order grammars and the safety constraint. Ph.D. thesis, University of Oxford (2006)

48. Muller, D.E., Schupp, P.E.: The theory of ends, pushdown automata, and second-order logic. Theoretical Computer Science 37, 51–75 (1985)
49. Murawski, A.S., Ong, C.H.L., Walukiewicz, I.: Idealized Algol with ground recursion and DPDA equivalence. In: ICALP. pp. 917–929 (2005)
50. Neatherway, R.P., Ramsay, S.J., Ong, C.H.L.: A traversal-based algorithm for higher-order model checking. In: ICFP. pp. 353–364 (2012)
51. Nivat, M.: Langages algébriques sur le magma libre et sémantique des schémas de programme. In: Proc. ICALP. pp. 293–308 (1972)
52. Nivat, M.: On the interpretation of recursive program schemes. Symposia Mathematica 15, 255–281 (1975)
53. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS. pp. 81–90 (2006), long version `www.cs.ox.ac.uk/people/luke.ong/personal/publications/lics06-long.pdf`.
54. Ong, C.H.L.: Local computation of beta-reduction by game semantics (2012), preprint, `www.cs.ox.ac.uk/people/luke.ong/personal/publications/locred.pdf`
55. Ong, C.H.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: POPL. pp. 587–598 (2011)
56. Parchmann, R., Duske, J., Specht, J.: On deterministic indexed languages. Information and Control 45, 48–67 (1980)
57. Park, D.M.R.: Fixpoint induction and proofs of program properties. In: Michie, D., Meltzer, B. (eds.) Machine Intelligence. vol. 5 (1970)
58. Parys, P.: Collapse operation increases expressive power of deterministic higher order pushdown automata. In: STACS. pp. 603–614 (2011)
59. Parys, P.: On the significance of the collapse operation. In: LICS. pp. 521–530 (2012)
60. Parys, P.: A pumping lemma for pushdown graphs of any level. In: STACS. pp. 54–65 (2012)
61. Patterson, M.: Equivalence problems in a model of computation. Ph.D. thesis, University of Cambridge (1967)
62. Plotkin, G.D.: LCF as a programming language. Theoretical Computer Science 5, 223–255 (1977)
63. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Maths. Soc 141, 1–35 (1969)
64. Salvati, S., Walukiewicz, I.: Krivine machines and higher-order schemes. In: ICALP (2). pp. 162–173 (2011)
65. Schwichtenberg, H.: Definierbare funktionen im lambda-kalkul mit typen. Archiv Logik Grundlagen-forsch 17 (1976)
66. T. Cachat: Games on Pushdown Graphs and Extensions. Ph.D. thesis, RWTH Aachen (2003), `http://www.liafa.jussieu.fr/~txc/Download/Cachat-PhD.pdf`
67. Walukiewicz, I.: Pushdown processes: games and model-checking. Information and Computation 157, 234–263 (2001)