# Higher-order pushdown trees are easy

Teodor Knapik[1], Damian Niwiński[2⋆], and Paweł Urzyczyn[2⋆⋆]

[1] Université de la Réunion, BP 7151,
97715 Saint Denis Messageries Cedex 9, Réunion
knapik@univ-reunion.fr
[2] Institute of Informatics, Warsaw University
ul. Banacha 2, 02–097 Warszawa, Poland
{niwinski,urzy}@mimuw.edu.pl

January 2002

**Abstract.** We show that the monadic second-order theory of an infinite tree recognized by a higher-order pushdown automaton of any level is decidable. We also show that trees recognized by pushdown automata of level $n$ coincide with trees generated by safe higher-order grammars of level $n$. Our decidability result extends the result of Courcelle on algebraic (pushdown of level 1) trees and our own result on trees of level 2.

## Introduction

The Rabin Tree Theorem, stating the decidability of the monadic second-order (MSO) theory of the full $n$-ary tree (SnS), is among the most widely applied decidability results. Rabin himself [15] inferred a number of decidability results for various mathematical structures interpretable in SnS (e.g., countable linear orders). Muller and Schupp [14] gave rise to the study of graphs definable in SnS, by showing decidability of the MSO theory of any graph generated by a pushdown automaton; this result was further extended by Courcelle [3] to equational graphs, and by Caucal [2] to prefix–recognizable graphs.

However, a more sophisticated use of the Rabin Tree Theorem allows to go beyond the structures directly interpretable in SnS. Indeed, Courcelle [3] established the decidability of the MSO theory of any algebraic tree, i.e., a tree generated by a context–free (algebraic) tree grammar. Such a tree can be also presented as a computation tree of a pushdown automaton. The interest in this kind of structures (and their theories) arose in recent years in the verification community, in the context of verification of infinite state systems (see [13] and references therein, and [18] particularly for the model-checking problem on pushdown trees).

Context-free grammars and pushdown automata can be viewed as the first level of an infinite hierarchy of higher-order grammars and higher-order pushdown automata. These hierarchies, introduced in the early eighties by Engelfriet [6], have been subsequently extensively studied, in particular by Damm [4], and Damm and Goerdt [5]. We find it natural to examine the computation trees of higher-order pushdown automata, as well as trees generated by higher-order grammars, from the point of view of the decidability of their MSO theories. Indeed, we would like to push the frontier of decidability in order to capture more complex trees.

In [11], we have made the first step in this direction by showing decidability of the MSO theory of trees generated by grammars of level two subject to an additional restriction on grammars which we called *safety*. This constraint is similar to the restriction to "derived types" in [4, 5]. It is still an open problem if this restriction is essential for decidability, or whether it really reduces the generating power of grammars. However, in the present paper we are able to state two results.

(1) After a natural generalization of the concept of safety, a tree generated by a safe grammar of any level enjoys a decidable MSO theory.

⋆ Partly supported by KBN Grant 8 T11C 027 20.
⋆⋆ Partly supported by KBN Grant 8 T11C 028 20.

(2) A tree is generated by a safe grammar of level $n$ if and only if it can be recognized by a pushdown automaton of level $n$.

Consequently (by (1) and the "if" part of (2)), the monadic second order theory of a tree recognized by a pushdown automaton of any level is decidable.

The "only if" part of (2) is of independent interest: it can be understood as an implementation of higher-order recursion by higher-order stack. This implementation is simpler than that of [5] for the following reasons: because we are using a simpler notion of a higher-order pushdown store, and because it works essentially for arbitrary safe grammars, without any specific normal form restrictions. (The few syntactic conditions we assume are merely for convenience.)

The question of decidability of MSO theories of higher-level trees has been recently investigated also by Courcelle and Knapik [10]. Following ideas of Damm [4], the authors study an operation of *evaluation* which, when applied to a tree (generated by a grammar) of level $n$, produces a tree of level $n+1$. They show that this evaluation operation preserves MSO decidability, and that any algebraic (level 1) tree can be obtained by evaluation of a regular tree. While multiple application of evaluation leads to trees of arbitrary high level, it is not clear if all trees generated by save grammars can be obtained in this way.

Problems related to ours were addressed by H. Hungar, who studied graphs generated by some specific higher-order graph grammars. He showed [7] decidability of the monadic second-order theory (S1S) of *paths* of such graphs (not the full MSO theory of graphs).

## 1 Preliminaries

Throughout the paper, the set of natural numbers is denoted $\omega$ and, for $n \in \omega$, the symbol $[n]$ abbreviates $\{1, \ldots, n\}$.

**Types.** We consider a set of types $\mathcal{T}$ constructed from a unique *basic* type $\mathbf{0}$: $\mathbf{0}$ is a type and, if $\tau_1, \tau_2$ are types, so is $(\tau_1 \to \tau_2) \in \mathcal{T}$. The operator $\to$ is assumed to associate to the right. Note that each type is of the form $\tau_1 \to \cdots \to \tau_n \to \mathbf{0}$, for some $n \geq 0$. A type $\mathbf{0} \to \cdots \to \mathbf{0}$ with $n+1$ occurrences of $\mathbf{0}$ is also written $\mathbf{0}^n \to \mathbf{0}$.

The level $\ell(\tau)$ of a type $\tau$ is defined by $\ell(\mathbf{0}) = 0$, and $\ell(\tau_1 \to \tau_2) = \max(1 + \ell(\tau_1), \ell(\tau_2))$. Thus $\mathbf{0}$ is the only type of level 0 and each type of level 1 is of the form $\mathbf{0}^n \to \mathbf{0}$ for some $n > 0$. A type $\tau_1 \to \cdots \to \tau_n \to \mathbf{0}$ is *homogeneous* (where $n \geq 0$) if each $\tau_i$ is homogeneous and $\ell(\tau_1) \geq \ell(\tau_2) \geq \ldots \geq \ell(\tau_n)$. For example a type $((\mathbf{0} \to \mathbf{0}) \to \mathbf{0}) \to (\mathbf{0} \to \mathbf{0}) \to (\mathbf{0} \to \mathbf{0} \to \mathbf{0}) \to \mathbf{0} \to \mathbf{0}$ is homogeneous, but a type $\mathbf{0} \to (\mathbf{0} \to \mathbf{0}) \to \mathbf{0}$ is not.

**Higher–order terms.** A *typed alphabet* is a set $\Gamma$ of symbols with types in $\mathcal{T}$. Thus $\Gamma$ can be also presented as a $\mathcal{T}$-indexed family $\{\Gamma_\tau\}_{\tau \in \mathcal{T}}$, where $\Gamma_\tau$ is the set of all symbols of $\Gamma$ of type $\tau$. We let the *type level* $\ell(\Gamma)$ of $\Gamma$ be the supremum of $\ell(\tau)$, such that $\Gamma_\tau$ is nonempty.

Given a typed alphabet $\Gamma$, the set $T(\Gamma) = \{T(\Gamma)_\tau\}_{\tau \in \mathcal{T}}$ of *applicative terms* is defined inductively, by

(1) $\Gamma_\tau \subseteq T(\Gamma)_\tau$;
(2) if $t \in T(\Gamma)_{\tau_1 \to \tau_2}$ and $s \in T(\Gamma)_{\tau_1}$ then $(ts) \in T(\Gamma)_{\tau_2}$.

Note that each applicative term can be presented in a form $Zt_1 \ldots t_n$, where $n \geq 0$, $Z \in \Gamma$, and $t_1, \ldots, t_n$ are applicative terms. We say that a term $t \in T(\Gamma)_\tau$ is of type $\tau$, which we also write $t : \tau$. A term $t : \tau$ is said to be of *level* $k$ iff $\tau$ is of level $k$. We adopt the usual notational convention that application associates to the left, i.e. we write $t_0 t_1 \ldots t_n$ instead of $(\cdots((t_0 t_1) t_2) \cdots) t_n$. For applicative terms $t, t_1, \ldots, t_m$, and symbols $z_1, \ldots, z_m$, of appropriate types, the term $t[z_1{:=}t_1, \ldots, z_k{:=}t_k]$ is defined as the result of simultaneous replacement in $t$ of $z_i$ by $t_i$, for $i = 1, \ldots, m$.

**Trees.** The free monoid generated by a set $X$ is written $X^*$ and the empty word is written $\varepsilon$. The length of word $w \in X^*$ is denoted by $|w|$.

A *tree* is any nonempty prefix–closed subset $T$ of $X^*$ (with $\varepsilon$ considered as the *root*). If $u \in T$, $x \in X$, and $ux \in T$ then $ux$ is an *immediate successor* of $u$ in $T$. For $w \in T$, the set $T.w = \{v \in X^* : wv \in T\}$ is the *subtree* of $T$ induced by $w$. Note that $T.w$ is also a tree, and $T.\varepsilon = T$.

Now let $\Sigma$ be a typed alphabet of level 1. A symbol $f$ in $\Sigma$ is of type $\mathbf{0}^n \to \mathbf{0}$, for $n \geq 0$, and can be viewed as a (first–order) function symbol of *arity* n. Let $T \subseteq \omega^*$ be a tree. A mapping $t{:}\, T \to \Sigma$ is called a $\Sigma$*–tree* provided that if $t(w) : \mathbf{0}^k \to \mathbf{0}$ then $w$ has exactly $k$ immediate successors which are $w1, \ldots, wk$ (hence $w$ is a leaf whenever $t(w) : \mathbf{0}$). The set of $\Sigma$–trees is written $T^{\infty}(\Sigma)$.

If $t{:}\, T \to \Sigma$ is a $\Sigma$–tree, then $T$ is called the *domain* of $t$ and denoted by $T = \mathcal{D}\mathrm{om}\ t$. For $v \in \mathcal{D}\mathrm{om}\ t$, the *subtree* of $t$ induced by $v$ is a $\Sigma$–tree $t.v$ such that $\mathcal{D}\mathrm{om}\ t.v = (\mathcal{D}\mathrm{om}\ t).v$, and $t.v(w) = t(vw)$, for $w \in \mathcal{D}\mathrm{om}\ t.v$. It is convenient to organize the set $T^{\infty}(\Sigma)$ into an algebra over the signature $\Sigma$, where for each $f \in \Sigma_{\mathbf{0}^n \to \mathbf{0}}$, the operation associated with $f$ sends an $n$–tuple of trees $t_1, \ldots, t_n$ onto the unique tree $t$ such that $t(\varepsilon) = f$ and $t.i = t_i$, for $i \in [n]$. Finite trees in $T^{\infty}(\Sigma)$ can be also identified with applicative terms of type $\mathbf{0}$ over the alphabet $\Sigma$ in the usual manner.

We also introduce a concept of limit. For a $\Sigma$-tree $t$, let $t{\restriction}n$ be its truncation to the level $n$, i.e., the restriction of the function $t$ to the set $\{w \in \mathcal{D}\mathrm{om}\ t \mid |w| \leq n\}$. Suppose $t_0, t_1, \ldots$ is a sequence of $\Sigma$–trees such that, for all $k$, there is an $m$, say $m(k)$, such that, for all $n, n' \geq m(k)$, $t_n{\restriction}k = t_{n'}{\restriction}k$. (This is a Cauchy condition in a suitable metric space of trees.) Then the *limit* of the sequence $t_n$, in symbols $\lim t_n$, is a $\Sigma$–tree $t$ which is the set-theoretical union of the functions $t_n{\restriction}m(n)$ (understanding a function as a set of pairs).

**Monadic second–order logic.** Let $R$ be a *relational vocabulary*, i.e., a set of relational symbols, each $r$ in $R$ given with an arity $\rho(r) > 0$. The formulas of *monadic second order (MSO) logic* over vocabulary $R$ use two kinds of variables : *individual variables* $x_0, x_1, \ldots$, and *set variables* $X_0, X_1, \ldots$. Atomic formulas are $x_i = x_j$, $r(x_{i_1}, \ldots, x_{i_{\rho(r)}})$, and $X_i(x_j)$. The other formulas are built using propositional connectives $\vee, \neg$, and the quantifier $\exists$ ranging over both kinds of variables. (The connectives $\wedge, \Rightarrow$, etc., as well as the quantifier $\forall$ are introduced in the usual way as abbreviations.) A formula without free variables is called a *sentence*. Formulas are interpreted in relational structures over the vocabulary $R$, which we usually present by $\mathbf{A} = \langle A, \{r^{\mathbf{A}} : r \in R\}\rangle$, where $A$ is the *universe* of $\mathbf{A}$, and $r^{\mathbf{A}} \subseteq A^{\rho(r)}$ is a $\rho(r)$-ary relation on $A$. A *valuation* is a mapping $v$ from the set of variables (of both kinds), such that $v(x_i) \in A$, and $v(X_i) \subseteq A$. The *satisfaction* of a formula $\varphi$ in $\mathbf{A}$ under the valuation $v$, in symbols $\mathbf{A}, v \models \varphi$ is defined by induction on $\varphi$ in the usual manner.

Clearly, the satisfaction of a formula depends only on the valuation of its free variables. The *monadic second–order theory* of $\mathbf{A}$ is the set of all MSO sentences satisfied in $\mathbf{A}$, in symbols $MSO(\mathbf{A}) = \{\varphi : \mathbf{A} \models \varphi\}$.

Let $\Sigma$ be a typed alphabet of level 1, and suppose that the maximum of the arities of symbols in $\Sigma$ exists and equals $m_\Sigma$. A tree $t \in T^{\infty}(\Sigma)$ can be viewed as a logical structure $\mathbf{t}$, over the vocabulary $R_\Sigma = \{p_f : f \in \Sigma\} \cup \{d_i : 1 \leq i \leq m_\Sigma\}$, with $\rho(p_f) = 1$, and $\rho(d_i) = 2$:

$$\mathbf{t} = \langle \mathcal{D}\mathrm{om}\, t,\ \{p_f^{\mathbf{t}} : f \in \Sigma\}\ \cup\ \{d_i^{\mathbf{t}} : 1 \leq i \leq m_\Sigma\}\rangle.$$

The universe of $\mathbf{t}$ is the domain of $t$, and the predicate symbols are interpreted by $p_f^{\mathbf{t}} = \{w \in \mathcal{D}\mathrm{om}\, t :\ t(w) = f\}$, for $f \in \Sigma$, and $d_i^{\mathbf{t}} = \{(w, wi) : wi \in \mathcal{D}\mathrm{om}\, t\}$, for $1 \leq i \leq m_\Sigma$. We refer the reader to [16] for a survey of the results on monadic second-order theory of trees.

**Grammars.** We now fix two disjoint typed alphabets, $N = \{N_\tau\}_{\tau \in \mathcal{T}}$ and $\mathcal{X} = \{\mathcal{X}_\tau\}_{\tau \in \mathcal{T}}$ of *nonterminals* and *variables* (or *parameters*), respectively. A *grammar* is a tuple $\mathcal{G} = (\Sigma, V, S, E)$, where $\Sigma$ is a *signature* (i.e., a finite alphabet of level 1), $V \subseteq N$ is a finite set of nonterminals, $\mathcal{S} \in V$ is a *start symbol* of type $\mathbf{0}$, and $E$ is a set of productions of the form

$$\mathcal{F} z_1 \ldots z_m \Rightarrow w$$

where $\mathcal{F} : \tau_1 \to \tau_2 \cdots \to \tau_m \to \mathbf{0}$ is a nonterminal in $V$, $z_i$ is a variable of type $\tau_i$, and $w$ is an applicative term in $T(\Sigma \cup V \cup \{z_1 \dots z_m\})$.

We assume that for each $\mathcal{F}$ in $V$, there is exactly one production in $E$ with $\mathcal{F}$ occurring on the left hand side. Furthermore, we make a *proviso* that each nonterminal in a grammar has a homogeneous type, and that if $m \geq 1$ then $\tau_m = \mathbf{0}$. This implies that each nonterminal of level $> 0$ has at least one parameter of level 0 (which needs not, of course, occur at the right-hand side). The *level* of a grammar is the highest level of its nonterminals.

In this paper, we are interested in grammars as generators of $\Sigma$-trees. Let, as before, $\Sigma^{\perp} = \Sigma \cup \{\perp\}$, with $\perp : \mathbf{0}$. First, with any applicative term $t$ over $\Sigma \cup V$, we associate an expression $t^{\perp}$ over signature $\Sigma^{\perp}$ inductively as follows.

- If $t = f$, $f \in \Sigma$, then $t^{\perp} = f$.
- If $t = X$, $X \in V$, then $t^{\perp} = \perp$.
- If $t = (sr)$ then if $s^{\perp} \neq \perp$ then $t^{\perp} = (s^{\perp} r^{\perp})$, otherwise $t^{\perp} = \perp$.

Informally speaking, the operation $t \mapsto t^{\perp}$ replaces in $t$ each nonterminal, together with its arguments, by $\perp$. It is easy to see that if $t$ is an applicative term (over $\Sigma \cup V$) of type $\mathbf{0}$ then $t^{\perp}$ is an applicative term over $\Sigma^{\perp}$ of type $\mathbf{0}$. Recall that applicative terms over $\Sigma^{\perp}$ of type $\mathbf{0}$ can be identified with finite trees.

We will now define the single-step rewriting relation $\to_{\mathcal{G}}$ among the terms over $\Sigma \cup V$. Informally speaking, $t \to_{\mathcal{G}} t'$ whenever $t'$ is obtained from $t$ by replacing some occurrence of a nonterminal $F$ by the right–hand side of the appropriate production in which all parameters are in turn replaced by the actual arguments of $F$. Such a replacement is allowed only if $F$ occurs as a head of a subterm of type $\mathbf{0}$. More precisely, the relation $\to_{\mathcal{G}} \subseteq T(\Sigma \cup V) \times T(\Sigma \cup V)$ is defined inductively by the following clauses.

- $\mathcal{F} t_1 \dots t_k \to_{\mathcal{G}} t[z_1{:=}t_1, \dots, z_k{:=}t_k]$ if there is a production $\mathcal{F} z_1 \dots z_k \Rightarrow t$ (with $z_i : \rho_i$, $i = 1, \dots, k$), and $t_i \in T(\Sigma \cup V)_{\rho_i}$, for $i = 1, \dots, k$.
- If $t \to_{\mathcal{G}} t'$ then $(st) \to_{\mathcal{G}} (st')$ and $(tq) \to_{\mathcal{G}} (t'q)$, whenever the expressions in question are applicative terms.

A *reduction* is a finite or infinite sequence of terms in $T(\Sigma \cup V)$, $t_0 \to_{\mathcal{G}} t_1 \to_{\mathcal{G}} \dots$. As usual, the symbol $\twoheadrightarrow_{\mathcal{G}}$ stands for the reflexive transitive closure of $\to_{\mathcal{G}}$. We also define the relation $t \twoheadrightarrow^{\infty}_{\mathcal{G}} t'$, where $t$ is an applicative term in $T(\Sigma \cup V)$ and $t'$ is a tree in $T^{\infty}(\Sigma^{\perp})$, by

- $t'$ is a finite tree, and there is a finite reduction sequence $t = t_0 \to_{\mathcal{G}} \dots \to_{\mathcal{G}} t_n = t'$, or
- $t'$ is infinite, and there is an infinite reduction sequence $t = t_0 \to_{\mathcal{G}} t_1 \to_{\mathcal{G}} \dots$ such that $t' = \lim t_n^{\perp}$.

To define a unique tree produced by the grammar, we recall a standard *approximation ordering* on $T^{\infty}(\Sigma^{\perp})$: $t' \sqsubseteq t$ if $\mathcal{D}om\ t' \subseteq \mathcal{D}om\ t$ and, for each $w \in \mathcal{D}om\ t'$, $t'(w) = t(w)$ or $t'(w) = \perp$. (In other words, $t'$ is obtained from $t$ by replacing some of its subtrees by $\perp$.) Then we let

$$[\![\mathcal{G}]\!] = \sup\{t \in T^{\infty}(\Sigma^{\perp}) : S \twoheadrightarrow^{\infty}_{\mathcal{G}} t\}$$

It is easy to see that, by the Church–Rosser property of our grammar, the above set is directed, and hence $[\![\mathcal{G}]\!]$ is well defined since $T^{\infty}(\Sigma^{\perp})$ with the approximation ordering is a cpo. Furthermore, it is routine to show that if an infinite reduction $S = t_0 \to_{\mathcal{G}} t_1 \to_{\mathcal{G}} \dots$ is *fair*, i.e., any occurrence of a nonterminal symbol is eventually rewritten, then its result $t' = \lim t_n^{\perp}$ is $[\![\mathcal{G}]\!]$.

If a tree $t$ is generated by a grammar of level $n$, we will sometimes say that $t$ is of level $n$.

## 2 Infinitary lambda calculus

In this section we recall some concepts and results from [11]. Our motivation for introducing infinite lambda terms comes from the strategy of our proof. Basically, we wish to reduce the MSO theory of a tree $t$ of level $n$ to the MSO theory of some tree $t'$ of level $n-1$. However, like in [11], it will be useful to view the latter tree as an infinite lambda term (intuitively, evaluating to $t$).

Infinitary lambda calculus is an extension of the ordinary lambda calculus, which allows the use of infinite lambda terms. We will identify infinite lambda terms with certain infinite trees. More specifically, we fix a *finite* alphabet $\Sigma$ of level 1 called *signature*, and let $\Sigma^{\perp} = \Sigma \cup \{\perp\}$, where $\perp$ is a fresh symbol of type $\mathbf{0}$. All our finite and infinite terms, called *lambda trees* are simply typed and may involve constants from $\Sigma^{\perp}$, and variables from a fixed countably infinite set. In fact, we only consider lambda trees of types of level at most 1.

**Definition 2.1.** Let $\Sigma^{\circ}$ be an infinite alphabet of level 1, consisting of

- A binary function symbol @;
- All symbols from $\Sigma^{\perp}$, as individual constants, regardless of their actual types;
- Infinitely many individual variables, as individual constants;
- Unary function symbols $\lambda x$ for all variables $x$.

The set of all *lambda trees* (over a signature $\Sigma$) is the greatest set of $\Sigma^{\circ}$-trees, given together with their *types*, such that the following conditions hold.

- Each variable $x$ is a lambda tree of type $\mathbf{0}$.
- Each function symbol $f \in \Sigma^{\perp}$ of type $\tau$ is a lambda tree of type $\tau$.
- Otherwise each lambda tree is of type of level at most 1 and is either an *application* $(MN)$ or an *abstraction* $(\lambda x.M)$ (see Figure 1).
- If a lambda tree $P$ of type $\tau$ is an application $(MN)$ then $M$ is a lambda tree of type $\mathbf{0} \to \tau$, and $N$ is a lambda tree of type $\mathbf{0}$.
- If a lambda tree $P$ of type $\tau$ has the form $(\lambda x.M)$, then $\tau = \mathbf{0} \to \sigma$, and $M$ is a lambda tree of type $\sigma$.

Strictly speaking, the above is a co-inductive definition of the two-argument relation "$M$ is a lambda tree of type $\tau$". Formally, a lambda tree can be presented as a pair $(M, \tau)$, where $M$ is a $\Sigma^{\circ}$-tree (as defined in section 1), and $\tau$ is its type satisfying the conditions above. Whenever we talk about a "lambda tree" we actually mean a lambda tree together with its type. We warn the reader about a possible confusion: The types associated with lambda trees by the above definition are *not* the types of these trees viewed as terms over $\Sigma^{\circ}$ (but rather as terms over $\Sigma$).

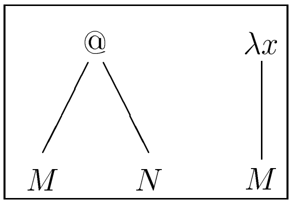


**Fig. 1.** Application and abstraction

Let $M$ be a lambda tree and let $x$ be a variable. Each node of $M$ labeled $x$ is called an *occurrence* of $x$ in $M$. An occurrence of $x$ is *bound*, iff it has an ancestor labeled $\lambda x$. The *binder* of this occurrence of $x$ is the closest of all such ancestors $\lambda x$ (i.e., one occurring at the largest depth). An occurrence of $x$ which is not bound is called *free*. We say that a variable $x$ is *free* in a lambda tree $M$ iff it has a free occurrence in $M$. The (possibly infinite) set of all free variables of $M$ will be denoted by $FV(M)$. A lambda tree $M$ with $FV(M) = \varnothing$ is called *closed*.

Clearly, ordinary lambda terms can be seen as a special case of lambda trees, and the notion of a free variable in a lambda tree generalizes the notion of a free variable in a lambda term. The *$n$-th approximant* of a lambda tree $M$, denoted $M{\restriction}n$ is defined by induction as follows:

- $M{\restriction}0 = \perp$, for all $M$;
- $(MN){\restriction}n{+}1 = (M{\restriction}n)(N{\restriction}n)$;
- $(\lambda x.M){\restriction}n{+}1 = \lambda x(M{\restriction}n)$.

That is, the $n$-th approximant is obtained by replacing all subtrees rooted at depth $n$ by the constant $\perp$.

Let $P(z/x)$ denote the result of replacing in $P$ all free occurrences of a variable $x$ by another variable $z$.

**Definition 2.2.** A *bisimulation* on lambda trees is a binary relation $\sim$ satisfying the following conditions:

(1) If $M \sim N$ then $M$ and $N$ are of the same type.
(2) If $M \sim N$ then the root labels of $M$ and $N$ are either the same or both $M$ and $N$ are abstractions.
(3) If $\lambda x.M \sim \lambda y.N$, then $M(z/x) \sim N(z/y)$, whenever $z$ is a fresh variable (neither free nor bound in $\lambda x.M$ and $\lambda y.N$).
(4) If $(MN) \sim (M'N')$ then $M \sim M'$ and $N \sim N'$.

**Lemma 2.3.** (1) *The union $\approx$ of all bisimulations is a bisimulation itself.*
(2) *On ordinary lambda-terms, the relation $\approx$ coincides with the $\alpha$-equivalence $=_\alpha$.*
(3) *$M \approx N$ holds if and only if $M{\restriction}n =_\alpha N{\restriction}n$ holds for all $n$.*
(4) *If $M \approx N$ then $FV(M) = FV(N)$.*

The above lemma provides evidence that the greatest bisimulation is a correct generalization of the notion of $\alpha$-equivalence. That is, $M \approx N$ holds iff $M$ and $N$ are identical up to names of bound variables. Following a common practice in lambda calculus, alpha-equivalent lambda trees will be identified. In consequence, all three statements $M \approx N$, $M =_\alpha N$ and $M = N$ will be understood as expressing the same property.

**Lemma 2.4 (Principle of co-induction [8]).** *In order to prove that $M =_\alpha N$, it is enough to find a bisimulation $\sim$ such that $M \sim N$.*

### 2.1 From grammar terms to lambda trees

Given a grammar $\mathcal{G}$, we define co-inductively a relation $\daleth_\mathcal{G}$ as the greatest relation between terms in $T(\Sigma \cup V \cup \mathcal{X})$ and lambda trees (over $\Sigma$) such that, for all $(t, M) \in \daleth_\mathcal{G}$,

(1) if $t$ is a function symbol $f$ then $M = f$,
(2) if $t$ is a variable $x \in \mathcal{X}_\mathbf{0}$ then $M = x$,
(3) if $t = \mathcal{F}t_1, \dots t_m$, where $\mathcal{F}$ is a nonterminal whose production in $\mathcal{G}$ is $\mathcal{F}\phi_1 \dots \phi_m x_1 \dots x_n \Rightarrow r$, with variables $\phi_1, \dots, \phi_m$ of level $\geq 1$ and variables $x_1, \dots, x_n$ of level 0, $t_1 \dots . t_m$ are terms such that $\mathrm{type}(\phi_i) = \mathrm{type}(t_i)$, for $i \in [m]$ then

$$M \in \{\lambda x_1' \dots x_n'.N \mid (r[\phi_1{:=}t_1, \dots, \phi_m{:=}t_m, x_1{:=}x_1', \dots, x_n{:=}x_n'], N) \in \daleth_\mathcal{G}\},$$

where the variables $x_1', \dots, x_n'$ are chosen so that no $x_i'$ occurs free in any of $t_j$,
(4) if $t = (t_1 t_2)$ where $t_1 : \mathbf{0} \to \tau$ and $t_2 : \mathbf{0}$ then $M \in \{(M_1 M_2) \mid (t_1, M_1), (t_2, M_2) \in \daleth_\mathcal{G}\}$.

It is easy to see that any term $t$ of level $\leq 1$ using variables only from $\mathcal{X}_\mathbf{0}$ is in the form (1)–(4).

**Lemma 2.5.** *$\daleth_\mathcal{G}$ is a partial function defined for all terms in $T(\Sigma \cup V \cup \mathcal{X}_\mathbf{0})$ of level $\leq 1$. That is, for any such $t$, there exists a unique $M$ (up to $\alpha$-equivalence) such that $(t, M) \in \daleth_\mathcal{G}$.*

From now on $\daleth_\mathcal{G}$ is considered as a function. The conditions (1)–(4) of the above definition can now be stated as follows.

(1) $\daleth_\mathcal{G}(t) = f$, if $t$ is a function symbol $f \in \Sigma$,
(2) $\daleth_\mathcal{G}(t) = x$, if $t$ is a variable $x \in \mathcal{X}_\mathbf{0}$,
(3) $\daleth_\mathcal{G}(t) = \lambda x_1' \dots x_n'.\daleth_\mathcal{G}(r[\phi_1{:=}t_1, \dots, \phi_m{:=}t_m, x_1{:=}x_1', \dots, x_n{:=}x_n'])$, if $t = \mathcal{F}t_1 \dots t_m$, for some nonterminal $\mathcal{F}$, such that $\mathcal{F}\phi_1 \dots \phi_m x_1 \dots x_n = r$ is a production of $\mathcal{G}$. The variables $x_1', \dots, x_n'$ are chosen so that no $x_i'$ occurs free in any of $t_j$,
(4) $\daleth_\mathcal{G}(t) = (\daleth_\mathcal{G}(t_1)\daleth_\mathcal{G}(t_2))$, if $t = (t_1 t_2)$ where $t_1 : \mathbf{0} \to \tau$ and $t_2 : \mathbf{0}$.

We will be mainly interested in $\daleth_\mathcal{G}(S)$, where $S$ is the start symbol of the grammar. It can be viewed as a lambda term obtained by a usually infinite process of expansion, according to the rules of the grammar. (Intuitively, it is a lambda term "evaluating" to the tree $[\![\mathcal{G}]\!]$.)

## 3 Safe grammars

The following is an extension of the definition from [11] to higher types.

**Definition 3.1.** A term of level $k > 0$ is *unsafe* if it contains an occurrence of a parameter of level strictly less than $k$, otherwise the term is *safe*. An *occurrence* of an unsafe term $t$ as a subexpression of a term $t'$ is *safe* if it is in the context $\dots(ts)\dots$, otherwise the occurrence is *unsafe*. A grammar is *safe* if no unsafe term has an unsafe occurrence at a right-hand side of any production.

**Example:** Let $f, g, h, a, b$ be signature symbols of arity 2,1,1,0,0, respectively. Consider a grammar of level 2 with nonterminals $S : \mathbf{0}$, and $\mathcal{F} : (\mathbf{0} \to \mathbf{0}) \to \mathbf{0} \to \mathbf{0} \to \mathbf{0}$, and productions

$$\mathcal{S} \Rightarrow \mathcal{F}gab$$

$$\mathcal{F}\varphi xy \Rightarrow f(\mathcal{F}(\mathcal{F}\varphi x)y(hy))(f(\varphi x)y)$$

This grammar is not safe, because of the unsafe subterm $\mathcal{F}\varphi x$, which is not applied to an argument. However, it is equivalent to an algebraic grammar with the following productions:

$$\mathcal{S} \Rightarrow \mathcal{G}(ga)b$$

$$\mathcal{G}zy \Rightarrow f(\mathcal{G}(\mathcal{G}zy)(hy))(fzy)$$

But life is not always so easy. If we replace the above production for $\mathcal{F}$ by a slightly different one:

$$\mathcal{F}\varphi xy \Rightarrow f(\mathcal{F}(\mathcal{F}\varphi x)y(hy))(f(\varphi y)x)$$

we obtain a grammar which is conjectured not to be equivalent to a safe grammar of any level.

The crucial observation is the following.

**Lemma 3.2.** *If a grammar $\mathcal{G}$ is safe then the lambda tree $\beth_{\mathcal{G}}(S)$ can be constructed using only the original variables of the grammar. That is, whenever clause (3) is applied, the variables $x'_1, \dots, x'_n$ can be chosen just $x_1, \dots, x_n$.*

*Proof.* We use safety only with respect to parameters of level $\mathbf{0}$. Observe that substitution preserves safety and a subterm of a safe term is again safe. This follows that whenever clause (3) is applied, the terms $t_1, \dots, t_m$ are safe. Formally, we modify the definition of $\beth_{\mathcal{G}}$ to $\beth'_{\mathcal{G}}$, such that the clause (3) applies only if the terms $t_1, \dots, t_m$ are safe (otherwise, say, $\beth'_{\mathcal{G}}(t) = \bot$), and show that $\beth'_{\mathcal{G}}(S)$ and $\beth_{\mathcal{G}}(S)$ are bisimilar using the condition (3) of Lemma 2.3. $\square$

In [11], we have essentially established the result which, by combining Theorem 2 and Proposition 4 from there with Lemma 3.2 above, can be rephrased as follows.[1]

**Theorem 3.3 ([11]).** *If a grammar $\mathcal{G}$ is safe then the MSO theory of the tree $[\![\mathcal{G}]\!]$ is reducible to the MSO theory of $\beth_{\mathcal{G}}(S)$, that is, there exists a recursive mapping of sentences $\varphi \mapsto \varphi'$ such that $[\![\mathcal{G}]\!] \models \varphi$ iff $\beth_{\mathcal{G}}(S) \models \varphi'$.*

By this, MSO theory of a tree generated by a safe grammar $\mathcal{G}$ reduces to the MSO theory of $\beth_{\mathcal{G}}(S)$. Our induction argument will consist in showing that the latter can be generated by a grammar of level $n - 1$.

[1] The main step in [11] is a reduction of the MSO theory of $[\![\mathcal{G}]\!]$ to the MSO theory of a certain graph associated with $\beth_{\mathcal{G}}(S)$ (Theorem 2 and Proposition 4 there). Lemma 3.2 allows to interpret that graph in the MSO theory of $\beth_{\mathcal{G}}(S)$.

**Definition 3.4.** Let $\mathcal{G} = (\Sigma, V, S, E)$ be a safe grammar of level $n \geq 2$. We may assume that the parameters of type $\mathbf{0}$ occurring in distinct productions are different. Let $\mathcal{X}^{\mathcal{G}_0} = \{x_1, \ldots, x_L\}$ be the set of all parameters of type $\mathbf{0}$ occurring in grammar $\mathcal{G}$. We define a grammar of level $n-1$ $\mathcal{G}^\alpha = (\Sigma^\alpha, V^\alpha, S^\alpha, E^\alpha)$ as follows.

It is convenient first to define a transformation $\alpha$ of (homogeneous) types, that maps $\mathbf{0}$ to $\mathbf{0}$, $\mathbf{0}^\ell \to \mathbf{0}$ to $\mathbf{0}$ and maps a type $\tau_1 \to \cdots \to \tau_n$ with $\ell(\tau_n) \leq 1$ and $\ell(\tau_{n-1}) > 1$, inductively, to $\alpha(\tau_1) \to \cdots \to \alpha(\tau_n)$. Note that, in particular, $\alpha$ maps $(\mathbf{0}^{k_1} \to \mathbf{0}) \to \cdots \to (\mathbf{0}^{k_m} \to \mathbf{0}) \to \mathbf{0}^\ell \to \mathbf{0}$ to $\mathbf{0}^m \to \mathbf{0}$. We will denote $\alpha(\tau)$ by $\tau^\alpha$. Let $\Sigma^\alpha = \Sigma \cup \{@, \lambda x_1, \ldots, \lambda x_L, x_1, \ldots, x_L\}$, where all symbols from $\Sigma$ as well as (former) parameters $x_1, \ldots, x_L$ are of arity 0, the symbol @ is binary, and the symbols $\lambda x_i$ are unary. The set $V^\alpha = \{\mathcal{F}^\alpha : \mathcal{F} \in V\}$ is a copy of $V$, with $\mathcal{F}^\alpha : \tau^\alpha$, whenever $\mathcal{F} : \tau$. Finally, whenever there is a production $\mathcal{F}\phi_1 \ldots \phi_m y_1 \ldots y_n \Rightarrow r$ in $E$, where $y_1, \ldots, y_n$ are the parameters of level 0, there is a production $\mathcal{F}^\alpha \phi_1 \ldots \phi_m \Rightarrow \lambda y_1 \ldots \lambda y_n . r^\alpha$ in $E^\alpha$, where the transformation of typed terms $r : \tau \mapsto r^\alpha : \tau^\alpha$ is defined inductively by

- $\alpha : \mathcal{F} \mapsto \mathcal{F}^\alpha$,
- $\alpha : z \mapsto z$, for any parameter $z$,
- $\alpha : (ts) \mapsto (t^\alpha s^\alpha)$, whenever $s : \tau$ with $\ell(\tau) \geq 1$,
- $\alpha : (ts) \mapsto ((@t^\alpha)s^\alpha)$, whenever $s : \mathbf{0}$ (hence consequently $t^\alpha, s^\alpha : \mathbf{0}$).

Note that all parameters of $\mathcal{G}$ of type $\tau$ become parameters of $\mathcal{G}^\alpha$ of type $\tau^\alpha$ except for the parameters of type $\mathbf{0}$ which become constants.

The following is an immediate consequence of the definition.

**Lemma 3.5.** *If $\mathcal{G}$ is safe then $\mathcal{G}^\alpha$ is safe, too.*

**Lemma 3.6.** *The trees $\rrbracket_{\mathcal{G}}(S)$ and $[\![\mathcal{G}^\alpha]\!]$ coincide.*

*Proof.* By Principle of co-induction 2.4, we check that these trees are bisimilar using the condition (3) of Lemma 2.3. ☐

**Theorem 3.7.** *Let $\mathcal{G}$ be a safe grammar of level $n$. Then the monadic theory of $[\![\mathcal{G}]\!]$ is decidable.*

*Proof.* For $n = 0$ the claim amounts to the decidability of the MSO theory of a regular tree that was essentially established already by Rabin [15]. For grammars of level 1 (algebraic), the result was proved by Courcelle [3]. (Clearly, for level $n \leq 1$, the safety assumption holds trivially.)

Now let $t = [\![\mathcal{G}]\!]$ be a tree generated by a grammar of level $n \geq 2$ and let $t^\alpha = [\![\mathcal{G}^\alpha]\!]$. By induction hypothesis, the MSO theory of $t^\alpha$ is decidable. The claim follows from Lemma 3.6 and Theorem 3.3. ☐

## 4 Pushdown automata

We use the abbreviation "pds" for the expression "pushdown store". A *level* 1 *pds* (or a 1-*pds*) over an alphabet $A$ is an arbitrary sequence $[a_1, \ldots, a_l]$ of elements of $A$, with $l \geq 1$. A *level n pds* (or a *n-pds*), for $n \geq 2$, is a sequence $[s_1, \ldots, s_l]$ of $(n-1)$-pds's, where $l \geq 1$. (That is, we assume that a pds is never empty.) For a given symbol $\perp \in A$, we define $\perp_k$ as follows: $\perp_1 = [\perp]$ and $\perp_{k+1} = [\perp_k]$. Thus $\perp_k$ is the level $k$ pds which contains only $\perp$ at the bottom.

Let $s$ be an $n$-pds and let $s'$ be a $k$-pds, for some $k \leq n$. We write $s' \subset s$ iff one of the following cases holds:

- $s = [s_1, \ldots, s_l]$ and $s' = s_i$, for some $i = 1, \ldots, l-1$.
- $s = [s_1, \ldots, s_l]$ and $s' \subset s_i$, for some $i = 1, \ldots, l$.

The following operations are possible on pushdown stores

- $push_1^a([a_1, \ldots, a_{l-1}, a_l]) = [a_1, \ldots, a_l, a]$, where $a \in A$;
- $pop_1([a_1, \ldots, a_{l-1}, a_l]) = [a_1, \ldots, a_{l-1}]$;

- $top_1([a_1, \ldots, a_{l-1}, a_l]) = a_l$.

On pushdown stores of level $n > 1$ one can perform the following operations:

- $push_n([s_1, \ldots, s_{l-1}, s_l]) = [s_1, \ldots, s_l, s_l]$;
- $pop_n([s_1, \ldots, s_{l-1}, s_l]) = [s_1, \ldots, s_{l-1}]$;
- $push_k([s_1, \ldots, s_{l-1}, s_l]) = [s_1, \ldots, s_{l-1}, push_k(s_l)]$, where $2 \leq k < n$;
- $push_1^a([s_1, \ldots, s_{l-1}, s_l]) = [s_1, \ldots, s_{l-1}, push_1^a(s_l)]$, where $a \in A$;
- $pop_k([s_1, \ldots, s_{l-1}, s_l]) = [s_1, \ldots, s_{l-1}, pop_k(s_l)]$, where $1 \leq k < n$;
- $top_n([s_1, \ldots, s_{l-1}, s_l]) = s_l$.
- $top_k([s_1, \ldots, s_{l-1}, s_l]) = top_k(s_l)$, for $1 \leq k < n$.

The operation $pop_k$ is undefined on a push down store, whose top pds of level $k$ consists of only one element.

Let $\Sigma$ be a signature, and let $Q$ and $A$ be finite sets. Let $\perp \in A$ be a distinguished element of $A$. The set $\mathcal{I}_n$ of *instructions* of level $n$ (parameterized by $\Sigma$, $Q$ and $A$) consists of all tuples of the following forms:

(1) $(push_k, p)$, where $p \in Q$ and $1 < k \leq n$;
(2) $(push_1^a, p)$, where $p \in Q$, and $a \in A$, $a \neq \perp$.
(3) $(pop_k, p)$, where $p \in Q$ and $1 \leq k \leq n$.
(4) $(f, p_1, \ldots, p_r)$, where $f \in \Sigma_r$ and $p_1, \ldots, p_r \in Q$;

A *pushdown automaton of level* $n$ is defined as a tuple

$$\mathcal{A} = \langle Q, \Sigma, A, q_0, \delta, \perp \rangle,$$

where $Q$ is a finite set of *states*, with an *initial state* $q_0$, $\Sigma$ is a signature, $A$ is a pds alphabet, with a distinguished *bottom symbol* $\perp$, and

$$\delta : Q \times A \ \to \mathcal{I}_n$$

is a *transition function*.

A *configuration* of an automaton $\mathcal{A}$ as above is a pair $\langle q, s \rangle$, where $q \in Q$ and $s$ is an $n$-pds over $A$. The *initial configuration* is $\langle q_0, \perp_n \rangle$. The set of all configurations is denoted by $\mathcal{C}$.

We define a relation $\to_{\mathcal{A}}$ on configurations as follows:

(1) If $top_1(s) = a$ and $\delta(q, a) = (push_k, p)$, with $k > 1$, then $\langle q, s \rangle \to_{\mathcal{A}} \langle p, push_k(s) \rangle$.
(2) If $top_1(s) = a$ and $\delta(q, a) = (push_1^b, p)$, then $\langle q, s \rangle \to_{\mathcal{A}} \langle p, push_1^b(s) \rangle$.
(3) If $top_1(s) = a$ and $\delta(q, a) = (pop_k, p)$, then $\langle q, s \rangle \to_{\mathcal{A}} \langle p, pop_k(s) \rangle$, provided $pop_k(s)$ is defined.

The symbol $\twoheadrightarrow_{\mathcal{A}}$ stands for the reflexive and transitive closure of $\to_{\mathcal{A}}$.

Now let $t : T \to \Sigma$ be a $\Sigma$-tree. A partial function $\varrho : T \to \mathcal{C}$ defined on an initial fragment of $T$ is called a *partial run* of $\mathcal{A}$ on $t$ iff the following conditions hold:

- $\langle q_0, \perp_n \rangle \twoheadrightarrow_{\mathcal{A}} \varrho(\varepsilon)$.
- If $w \in T$ and $\varrho(w) = \langle q, s \rangle$ then $\delta(\langle q, top_1(s) \rangle) = (f, p_1, \ldots, p_r)$, where $t(w) = f \in \Sigma_r$ and $p_1, \ldots, p_r \in Q$. In addition, $\langle p_i, s \rangle \twoheadrightarrow_{\mathcal{A}} \varrho(wi)$, for each $i = 1, \ldots, r$ when $\varrho(wi)$ is defined.

If a partial run is total, it is called a *run*. If $\mathcal{A}$ has a run on $t$, then we say that $t$ is *accepted* by $\mathcal{A}$. It should be clear that any given automaton $\mathcal{A}$ can accept at most one tree.

**Remark:** The above definition of a pushdown automaton is based on the definitions from [12, 17, 1] rather than the original definition of Engelfriet, used in [6, 5]. The latter differs from ours in that a pushdown store of level $k$, for $k > 1$, is defined as a sequence of pairs $[(s_1, a_1), \ldots, (s_l, a_l)]$, where the $s_i$ are pds's of level $k - 1$, and the $a_i$ are symbols from the alphabet. An automaton has access to the symbol $a_l$, as well as to the top symbol of $s_l$, the top symbol of the top pds of $s_l$, etc. However, it is not difficult to see that these two models are equivalent. Indeed, the additional label of a $k$-pds can be stored on the top of its top 1-pds. (Note that a 1-pds on top of a $k$-pds is also on top of pds's of level $2, 3, \ldots, k-1$, and thus must carry up to $k$ additional labels.) Each move of an Engelfriet style automaton is then simulated by a sequence of moves of our automaton.

It should also be clear that our higher-order pushdown trees generalize to higher levels the pushdown trees considered by Walukiewicz in [18]. In the context of verification, pushdown automata (in general, nondeterministic) are considered as processes rather than acceptors, i.e., the input alphabet is omitted. The interest is focused on the graph of all possible configurations of a process. However, by suitable choice of a signature, it is easy to identify the tree of configurations of a higher-order pushdown process with the tree recognized by an automaton in our sense. The branching nodes in the tree correspond to the points of nondeterministic choice of the process.

## 5 Automata and grammars

**Theorem 5.1.** *Let $t$ be accepted by a pushdown automaton of level $n$. Then it is generated by a safe grammar of level $n$.*

**Corollary 5.2.** *The MSO theory of every tree recognized by a higher-order pushdown automaton is decidable.*

**Theorem 5.3.** *A tree generated by a safe grammar of level $n$ is accepted by a pushdown automaton of level $n$.*

### 5.1 Proof of Theorem 5.1

Assume that our automaton $\mathcal{A} = \langle Q, \Sigma, A, q_0, \delta, \perp \rangle$ has $m$ states $0, \ldots, m-1$. (It is convenient to simply identify the states with their numbers.) We use the following abbreviations: $\mathbf{1} = \mathbf{0}^m \to \mathbf{0}$, $\mathbf{2} = \mathbf{1}^m \to \mathbf{1}$, and so on: $\mathbf{k} = (\mathbf{k\text{-}1})^m \to \mathbf{k}$, up to $\mathbf{n} = (\mathbf{n\text{-}1})^m \to \mathbf{n\text{-}1}$. Observe that $\mathbf{k} = (\mathbf{k\text{-}1})^m \to (\mathbf{k\text{-}2})^m \to \cdots \to \mathbf{1}^m \to \mathbf{0}^m \to \mathbf{0}$, for each $k = 0, \ldots, n$.

We construct a grammar which generates the tree $t$, with the following nonterminals:

- For each $q \in Q$ and each $a \in A$, there is a nonterminal $\mathcal{F}_q^a$ of type $\mathbf{n}$.
- For each $k = 1, \ldots, n$ there is a nonterminal $\mathit{Void}_k$ of type $\mathbf{n\text{-}k}$.
- And there is an initial nonterminal $\mathcal{S}$ of type $\mathbf{0}$.

The initial production of our grammar is:

$$\mathcal{S} \Rightarrow \mathcal{F}_{q_0}^{\perp} \ \overrightarrow{\mathit{Void}}_1 \, \overrightarrow{\mathit{Void}}_2 \, \ldots \, \overrightarrow{\mathit{Void}}_n,$$

where $\overrightarrow{\mathit{Void}}_k$ stands for $m$ repetitions of $\mathit{Void}_k$. Other productions apply to the nonterminals $\mathcal{F}_i^a$ and depend on $\delta(q_i, a)$. In order to understand the productions below, one should interpret an expression of the form

$$\mathcal{F}_q^a \, \vec{x}_1 \vec{x}_2 \ldots \vec{x}_n$$

as a representation of a configuration $\langle q, s \rangle$, where $\mathit{top}_1(s) = a$, and each vector $\vec{x}_k = x_k^0 \ldots x_k^{m-1}$ refers to the possible configurations to which the automaton may return after executing $\mathit{pop}_k$. There is always $m$ such possible configurations, corresponding to the $m$ internal states. (A call to $\mathit{Void}_k$ corresponds to an attempt to pop an "empty" stack.) Observe that the types of variables in $\vec{x}_k$ are more complex for lower level pds's. We think of it as follows: the information contained in a parameter $x_k^i : \mathbf{n\text{-}k}$ occurring in the sequence $\vec{x}_k$ represents directly the contents of $\mathit{top}_k(s)$. The actual state of the $n$-pds should then be seen as a function depending on the contents of $\mathit{pop}_k(s)$, understood as a sequence of pds's of levels $k+1, k+2, \ldots, n$. Here are the productions:

- $\mathcal{F}_q^a \, \vec{x}_1 \vec{x}_2 \ldots \vec{x}_n \Rightarrow \mathcal{F}_p^a \, \vec{x}_1 \ldots \vec{x}_{k-1} \, (\mathcal{F}_0^a \, \vec{x}_1 \ldots \vec{x}_k) \ldots (\mathcal{F}_{m-1}^a \, \vec{x}_1 \ldots \vec{x}_k) \, \vec{x}_{k+1} \ldots \vec{x}_n$, if $\delta(q, a) = (\mathit{push}_k, p)$, with $k > 1$.

- $\mathcal{F}^a_q \vec{x}_1 \vec{x}_2 \ldots \vec{x}_n \Rightarrow \mathcal{F}^b_p(\mathcal{F}^a_0 \vec{x}_1)\ldots(\mathcal{F}^a_{m-1} \vec{x}_1) \vec{x}_2 \ldots \vec{x}_n$,
  if $\delta(q,a) = (push^b_1, p)$.
- $\mathcal{F}^a_q \vec{x}_1 \vec{x}_2 \ldots \vec{x}_n \Rightarrow x^p_k \vec{x}_{k+1} \ldots \vec{x}_n$,
  if $\delta(q,a) = (pop_k, p)$.
- $\mathcal{F}^a_q \vec{x} \Rightarrow f(\mathcal{F}^a_{p_1} \vec{x})\ldots(\mathcal{F}^a_{p_r} \vec{x})$,
  if $\delta(q,a) = (f, p_1, \ldots, p_r)$, where $\vec{x}$ stands for $\vec{x}_1 \vec{x}_2 \ldots \vec{x}_n$.

Note that, in particular, the production corresponding to a $pop_n$ is simply $\mathcal{F}^a_q \ldots \Rightarrow x^p_n$. Also note that the maximal incomplete applications of level $k$ at the right hand side do not contain variables of level less than $k$. It follows that our grammar is safe.

In order to show the correctness of the simulation we must make precise how an expression of type **n-k** should represent a $k$-pds. More precisely, we define terms $Code_{q,s}$ meant to represent the contents of $s$ in state $q$. If $s$ is an $n$-pds then $Code_{q,s}$ represents the whole configuration. The definition is by induction with respect to the length of the pds.

We begin with $k = 1$. If $s = [\bot]$, then $Code_{q,s} = \mathcal{F}^{\bot}_q \ \vec{Void}_1$, and if $s = push^a_1(s')$ then $Code_{q,s} = \mathcal{F}^a_q Code_{1,s'} \ldots Code_{m,s'}$.

For $k > 1$, and $s = [s_1]$ we define $Code_{q,s} = Code_{q,s_1} \ \vec{Void}_k$. If $s = [s_1, \ldots, s_l]$, with $l > 1$, then $Code_{q,s} = Code_{q,s_l} Code_{1,s'} \ldots Code_{m,s'}$, where $s' = [s_1, \ldots, s_{l-1}]$.

**Lemma 5.4.** *Let $\langle q, s \rangle \to_{\mathcal{A}} \langle p, s' \rangle$. Then $Code_{q,s} \to_{\mathcal{G}} Code_{p,s'}$.*

*Proof.* By inspection of the possible cases. □

Theorem 5.1 now follows from the following fact:

**Lemma 5.5.** *Let $\varrho$ be the run of $\mathcal{A}$ on $t : T \to \Sigma$ and let $w \in T$. Let $\varrho(w) = \langle q, s \rangle$, with $top_1(s) = a$, and let $\delta(\langle q, a \rangle) = (f, p_1, \ldots, p_r)$. Then $\mathcal{S} \twoheadrightarrow_{\mathcal{G}} t'$, for some finite tree $t'$ with $t'(w) = Code_{q,s}$.*

*Proof.* Induction with respect to the length of $w$. □

### 5.2 Proof of Theorem 5.3

The proof is based on an idea from [9], where it was shown how to implement recursive programs with level 2 procedures with help of a 2-pds. The simulation of [9] was possible at level 2 because the individual parameters were passed *by value*. Thus, a nonlocal access to an individual meant an immediate access to a register value and did not require any additional evaluation. Under the safety restriction, one can generalize the construction in [9] to all levels.

Suppose a safe grammar $\mathcal{G}$ of level $n$ generates $t : T \to \Sigma$. With no loss of generality we may assume the following:

- Whenever a right hand side of a production in $\mathcal{G}$ is of the form $fu_1 \ldots u_r$, where $f \in \Sigma_r$, then each of the terms $u_1, \ldots, u_r$ begins with a nonterminal or a variable (but not with a signature constant).
- The initial nonterminal $\mathcal{S}$ does not occur at the right hand sides of productions.

In addition, we assume that the formal parameters of any $m$-ary nonterminal $\mathcal{F}$ are always $x_1, \ldots, x_m$.

For each expression $u$ we define *formal parameters of* $u$, not to be confused with variables actually occurring in $u$.

- If $u$ begins with a nonterminal $\mathcal{F} : \sigma$, then the formal parameters of $u$ are the formal parameters of $\mathcal{F}$.
- If $u$ begins with a variable or a signature constant of type $\sigma = \tau_1 \to \cdots \to \tau_d \to \mathbf{0}$ then the formal parameters of $u$ are $x_i : \tau_i$, for $i \leq d$.

In addition, with every expression $u = F \ldots$, we associate a *formal operator*, which is a variable *head* of the same type as $F$.

We construct a level $n$ automaton $\mathcal{A}$ accepting $t$. The pushdown alphabet of $\mathcal{A}$ consists of all safe subexpressions of all the right hand sides of the productions of $G$. Every element of $\mathcal{A}$ represents a possible call to a nonterminal, or to a variable, together with a (possibly incomplete) list of actual parameters. In order to distinguish between identifiers of $\mathcal{G}$ and pds symbols, the latter will be called *items*.

The bottom pds symbol is $\mathcal{S}$, the initial nonterminal. The set of states includes $q_0, q_1, \ldots q_g$, where $g$ is the maximal arity of an identifier[2] occurring in the grammar. The intended meaning of these states is as follows: in a configuration $\langle q_0, s \rangle$ the automaton attempts to evaluate the expression $top_1(s)$, while in a configuration $\langle q_i, s \rangle$ with $i > 0$ and $top_1(s) = Fu_1u_2 \ldots$ the goal is to evaluate the $i$-th argument of $F$. The automaton works in phases beginning and ending in the distinguished states $q_i$, using some auxiliary states in between.

We now describe the possible behaviour in each phase, assuming the following induction hypothesis to hold for every configuration $\langle q_i, s \rangle$.

- All the "internal" items (not on top of 1-pds's) begin with nonterminals.
- If an item $t$ occurs on a 1-pds directly atop of another item of the form $\mathcal{F} \ldots$ then all variables occurring in $t$ are formal parameters of $\mathcal{F}$.
- If $i \neq 0$ then $top_1(s)$ has at least $i$ formal parameters.
- Let $s' \subset s$ be a pds of level $n-k$, and let $s''$ be a $(n-k)$-pds occurring directly atop $s'$ on the same $(n-k+1)$-pds. Then
  - $top_1(s')$ is an expression beginning with a variable $\varphi : \tau$ of level $k$.
  - If $i \neq 0$ and $s'' = top_{n-k+1}(s)$ and the $i$-th formal parameter $x_i$ of $top_1(s)$ is of level $k$ then $x_i$ has type $\tau$.
  - Otherwise, the topmost incomplete application on $s''$ of level $k$ has type $\tau$.

Assume that the current configuration of the automaton is $\langle q_0, s \rangle$.
**Case 1:** Let $top_1(s) = \mathcal{F}u_1 \ldots u_d$ where $\mathcal{F}$ is a nonterminal and $\mathcal{F}x_1 \ldots x_n \Rightarrow t$ is the corresponding production. Then the automaton executes the instruction $push_1^t$, and remains in state $q_0$.
**Case 2:** If $top_1(s) = f$ or $top_1(s) = ft_1 \ldots t_r$ with $f \in \Sigma_r$ then the next instruction is $(f, q_1, \ldots q_r)$.
**Case 3:** If $top_1(s) = x : \mathbf{0}$ then $x$ must be a formal parameter of the previous item $top_1(pop_1(s))$, say the $j$-th one. The automaton executes $(pop_1, q_j)$.
**Case 4:** Let $top_1(s) = \varphi t_1 \ldots t_d$, where $\varphi$ is a variable of level $k > 0$ and arity at least $d$. Assume that $\varphi$ is an $r$-th formal parameter of $top_1(pop_1(s))$. The actions to be executed are $push_{n-k+1}$ followed by $pop_1$, and the next state is $q_r$.
That is, a call to a variable of level $1, 2, \ldots, n-1$ results in a "push" operation respectively of level $n, n-1, \ldots, 2$. The higher is the level of the variable, the lower is the level of the corresponding pds operation.
Now consider a current configuration of the automaton of the form $\langle q_i, s \rangle$, with $i > 0$.
**Case 5:** If $top_1(s) = Ft_1 \ldots t_r$ and $i \leq r$ then the top item is simply replaced by $t_i$ and the next state is $q_0$.
**Case 6:** Let $top_1(s) = Ft_1 \ldots t_r$, but $i > r$. Assume that $top_1(s) : \tau$ is of level $k$. The action to be performed is $(pop_{n-k+1}, q_{i-r})$.

Now we prove the correctness of this construction. First we define the *meaning* of an expression $u$ at $s$, written as $[\![u]\!]_s$.

- If $F$ is a signature constant or a nonterminal, then $[\![F]\!]_s = F$.
- If $u$ is an application $t_1t_2$ then $[\![u]\!]_s = [\![t_1]\!]_s[\![t_2]\!]_s$.
- If $top_1(s) = Ft_1 \ldots t_r$, then $[\![head]\!]_s = [\![F]\!]_{pop_1(s)}$. An exception is when $top_1(s) = \mathcal{S}$, in which case we set $[\![head]\!]_s = \mathcal{S}$.
- Let $top_1(s) = Ft_1 \ldots t_r$, and let $x_d$ be the $d$-th formal parameter at $top_1(s)$. If $d \leq r$ then $[\![x_d]\!]_s = [\![t_d]\!]_{pop_1(s)}$.

[2] Variables, signature constants and nonterminals are all called *identifiers*. An identifier of type $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \mathbf{0}$ is said to be of *arity* $n$.

- Otherwise, $[\![x_d]\!]_s = [\![x_{d-r}]\!]_{pop_{n-k+1}(s)}$, where $k$ is the level of $Ft_1 \dots t_r$.

If $top_1(s) = Ft_1 \dots t_r$, where $F$ is an identifier of arity $m \geq r$, then we define $[\![s]\!] = [\![head x_1 \dots x_r]\!]_s$.

**Lemma 5.6.** *For an arbitrary identifier $x$ of level $k = 1, \dots, n-1$, we have*

$$[\![x]\!]_s = [\![x]\!]_{top_{n-k+1}(s)}$$

*In other words, the meaning of $x$ is determined by the top pds of level $n-k$.*

*Proof.* Induction with respect to $s$. Note that to define $[\![x]\!]_s$ one never has to consider variables of levels less than $k$. This is because the grammar $\mathcal{G}$ is assumed to be safe. □

**Lemma 5.7.** *Assume that the configuration $\langle q_i, s \rangle$ with $i > 0$ satisfies our induction hypothesis. Then $\langle q_i, s \rangle \twoheadrightarrow_{\mathcal{A}} \langle q_0, s' \rangle$, for some $s'$ with $[\![s']\!] = [\![x_i]\!]_s$.*

*Proof.* The proof is by induction with respect to $s$. If the automaton moves according to Case 5, the hypothesis is immediate. Otherwise we have $[\![x_r]\!]_s = [\![x_{i-r}]\!]_{pop_{n-k+1}(s)}$ and we apply induction. □

**Lemma 5.8.** *Assume that the configuration $\langle q_0, s \rangle$ satisfies our induction hypothesis, and let the top item $top_1(s)$ begin with a variable. Then $\langle q_0, s \rangle \twoheadrightarrow_{\mathcal{A}} \langle q_0, s' \rangle$, where $[\![s']\!] = [\![s]\!]$ and $top_1(s)$ begins with a nonterminal or a signature constant.*

*Proof.* If the head variable $\varphi$ is of level $k$ then we have $[\![\varphi]\!]_{pop_1(s)} = [\![\varphi]\!]_{pop_1(push_{n-k+1}(s))}$, by Lemma 5.6. Thus the actions performed according to Case 4 do not change the meaning of the pds. Termination follows by induction on $n-k$ and the size of $top_{n-k+1}(s)$. □

**Lemma 5.9.** *Assume that the configuration $\langle q_0, s \rangle$ satisfies our induction hypothesis, and that $[\![s]\!] = t$. Suppose that $t \to_G t_1 \to_G \cdots \to_G t_m$ and let $t_m = gu_1 \dots u_r$ be the first term in the reduction sequence which begins with a signature constant. Then $\langle q_0, s \rangle \twoheadrightarrow_{\mathcal{A}} \langle q_0, s' \rangle$, where $[\![s']\!] = gu_1 \dots u_r$.*

*Proof.* The proof is by induction with respect to the number of steps in the reduction sequence $t \twoheadrightarrow_G gu_1 \dots u_r$. There are cases depending on the first reduction step $t \to_G t_1$. The nontrivial case is when $top_1(s)$ begins with a variable $\varphi$, i.e., not with the same symbol as $t$. This corresponds to Case 4 in the definition of the automaton. But it follows from Lemma 5.8 that the meaning $[\![s]\!]$ remains unchanged until a head nonterminal or terminal is exposed. □

**Lemma 5.10.** *Suppose that $\mathcal{S} \to_G t_1 \to_G \cdots \to_G t_m$ is an outermost reduction sequence and that $t_m$ is the first in the sequence term with $t_m(w) = f$, where $f$ is a signature constant. Let the subterm of $t_m$ occurring at $w$ be $fu_1 \dots u_k$. Then $\mathcal{A}$ has a partial run $\varrho$ on $t$ such that $\varrho(w) = \langle q_0, s \rangle$, with $top_1(s) = fv_1 \dots v_k$ and $[\![s]\!] = fu_1 \dots u_k$.*

*Proof.* Induction with respect to the length of $w$. Assuming induction hypothesis for the immediate predecessor of $w$, the automaton then simulates grammar reductions. First, a "split" action is executed without changing the pds, and we apply Lemma 5.7. Then we evaluate the expression on top of the pds and the correctness of this evaluation follows from Lemma 5.9. □

The conclusion is that for every $w \in \mathcal{D}\mathrm{om}\, t$ there is a partial run that reaches $w$. Thus the unique tree accepted by $\mathcal{A}$ coincides with $t$.

## References


[1] J. Tiuryn and A.J. Kfoury and P. Urzyczyn. On the expressive power of finitely typed and universally polymorphic recursive procedures. *Theoretical Comput. Sci.*, 93:1–41, 1992.

[2] D. Caucal. On infinite transition graphs having a decidable monadic second–order theory. In F. Meyer auf der Heide and B. Monien, editors, *23th International Colloquium on Automata Languages and Programming*, LNCS 1099, pages 194–205, 1996. A long version will appear in TCS.

[3] B. Courcelle. The monadic second–order theory of graphs IX: Machines and their behaviours. *Theoretical Comput. Sci.*, 151:125–162, 1995.

[4] W. Damm. The IO– and OI–hierarchies. *Theoretical Comput. Sci.*, 20(2):95–208, 1982.

[5] W. Damm and A. Goerdt. An automata-theoretic characterization of the oi-hierarchy. *Information and Control*, 71:1–32, 1986.

[6] J. Engelfriet. Iterated push-down automata and complexity classes. In *Proc. 15th STOC*, pages 365–373, 1983.

[7] H. Hungar. Model checking and higher-order recursion. In L. Pacholski and M. Kutyłowski and T. Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999*, LNCS 1672, pages 149–159, 1999.

[8] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of EATCS*, 1997(62):222–259.

[9] A. Kfoury and P. Urzyczyn. Finitely typed functional programs, part ii: comparisons to imperative languages. Report, Boston University, 1988.

[10] B. Courcelle and T. Knapik. The evaluation of first–order substitution is monadic second–order compatible. *Theoretical Comput. Sci.*, 2002. To appear.

[11] T. Knapik, D. Niwiński, and P. Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In *Typed Lambda Calculi and Applications, 5th International Conference*, LNCS 2044, pages 253–267. Springer-Verlag, 2001.

[12] W. Kowalczyk, D. Niwiński, and J. Tiuryn. A generalization of of cook's auxiliary-pushdown-automata theorem. *Fundamenta Informaticae*, 12:497–506, 1989.

[13] O. Kupferman and M. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Computer Aided Verification, Proc. 12th Int. Conference*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[14] D. Muller and P. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Comput. Sci.*, 37:51–75, 1985.

[15] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Soc*, 141:1–35, 1969.

[16] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer-Verlag, 1997.

[17] J. Tiuryn. Higher-order arrays and stacks in programming: An application of complexity theory to logics of programs. In *Proc. 12th MFCS*, LNCS 233, pages 177–198. Springer-Verlag, 1986.

[18] I. Walukiewicz. Pushdown processes: Games and model checking. *Information and Computation*, 164(2):234–263, 2001.