Simply typed fixpoint calculus and collapsible pushdown automata

SYLVAIN SALVATI and IGOR WALUKIEWICZ

Mathematical Structures in Computer Science / *FirstView* Article / May 2015, pp 1 - 47
DOI: 10.1017/S0960129514000590, Published online: 18 March 2015

Math. Struct. in Comp. Science: page 1 of 47.
doi:10.1017/S0960129514000590

Simply typed fixpoint calculus and collapsible pushdown automata

SYLVAIN SALVATI and IGOR WALUKIEWICZ[†]

INRIA, CNRS, Université de Bordeaux, France

Received 30 October 2012; revised 5 November 2014

Simply typed λ-calculus with fixpoint combinators, λY-calculus, offers an interesting method for approximating program semantics. The Böhm tree of a λY-term represents the meaning of the program up to the meaning of built-in constants. It is much easier to reason about properties of such trees than properties of interpreted programs. Moreover, some interesting properties of programs are already expressible on the level of these trees.

Collapsible pushdown automata (CPDA) give another way of generating the same class of trees as λY-terms. We clarify the relationship between the two models. In particular, we present two relatively simple translations from λY-terms to CPDA using Krivine machines as an intermediate step. The latter are general machines for describing computation of the weak head normal form in the λ-calculus. They provide the notions of closure and environment that facilitate reasoning about computation.

1. Introduction

Terms of simply typed λ-calculus with fixpoint combinators, λY-calculus, are abstract forms of programs where the meaning of constants is not specified. In consequence, the meaning of a term is its Böhm tree: a potentially infinite tree labelled with constants obtained from the evaluation of the term. In the context of this paper, Böhm trees are much more appropriate than normal forms, since due to the free interpretation of constants λY-terms representing programs rarely have a normal form. CPDA is another, more recent model with the same generating power. The main contribution of the paper are two translations from λY-terms to CPDA that clarify the relation between the two formalisms. Our translations show how to implement to standard notions of functional programming, *environments* and *closures*, with higher-order stacks and the *collapse* operation. The translations also explain the behaviour of the so-called safe fragment of the λY-calculus that can be translated to CPDA not using the *collapse* operation.

Ianov (1969) has introduced recursive schemes as a means for studying program transformation and control structures. Such schemes correspond to λY-terms of order 0. In a series of papers (Nivat 1972a,b), Nivat studies *algebraic* recursive schemes that correspond to λY-terms of order 1. The study of recursion on higher types as a control structure for programming languages was started in Milner (1973) and Plotkin (1977). Higher-order features allow for compact high-level programs. They have been present since

[†] Supported by ANR 2010 BLAN 0202 01 FREC

the beginning of programming, and appear in modern programming languages like Java 8, C++, Haskell, Javascript, Python, or Scala. Higher-order features allow one to write code that is closer to the specification and, consequently, more reliable. This is particularly useful in contexts where high assurance should come together with very complex functionality. Telephone switches, simulators, translators, statistical programs operating on terabytes of data, have been successfully implemented using functional languages[†].

The Böhm tree semantics of λY-terms is an informative intermediate step in giving a denotational semantics of a program. The meaning of a program can be obtained by considering it as a λY-term, taking the Böhm tree of the term, and applying a homomorphism giving a meaning to each of the constants. Yet, in some cases the Böhm tree gives already interesting information about the program. For example, resource usage patterns can be formulated in fragments of monadic second-order logic and verified over such trees (Kobayashi 2013). This is possible thanks to the fact that MSOL model checking is decidable for trees generated by higher-order recursive schemes (Ong 2006), and consequently for Böhm trees of λY-terms.

The study of Böhm trees generated by λY-terms has been mostly done by the language theory community. Following the tradition initiated by Ianov, these trees were studied under the form of recursive schemes rather than λY-terms. Damm (1982) has shown that considered as word generating devices, a class of so-called safe schemes is equi-expressive with the higher-order indexed languages introduced by Aho (1968) and Maslov (1974). These languages in turn have been known to be equivalent to the higher-order pushdown automata of Maslov (Damm and Goerdt 1986; Maslov 1976). Later it has been shown that trees generated by higher-order safe schemes are the same as those generated by higher-order pushdown automata (Knapik *et al.* 2002). This gave rise to the so-called pushdown hierarchy (Caucal 2002) and its numerous characterizations (Carayol and Wöhrle 2003).

The safety restriction has been tackled much more recently. First, because it has been somehow implicit in Damm (1982), and only brought on the front stage by Knapik *et al.* (2002). Second, because it required new insights in the nature of higher-order computation. Pushdown automata have been extended with a so-called panic operation (Aehlig *et al.* 2005; Knapik *et al.* 2005). This lead to the characterization of trees generated by schemes of order one. Later this operation has been extended to all higher-order stacks, and has been renamed collapse. Higher-order stack automata with collapse (CPDA) characterize recursive schemes at all orders (Hague *et al.* 2008). The fundamental question whether the collapse operation adds expressive power has been answered affirmatively only very recently (Parys 2012): there is a tree generated by an order 1 scheme that cannot be generated by a higher-order stack automaton without collapse.

The two formalisms, λY-calculus and CPDA, are equi-expressive but very different. In λY-calculus we have the notion of computation given by $\alpha\beta\delta$-reduction. Such a computation can be unbounded because reductions may make the term grow, but also, and this is very important, because α-conversion may introduce an unbounded number of fresh variables during computation. CPDA come with an explicit notion of state and

[†] For some examples see 'Functional programming in the real world' `http://homepages.inf.ed.ac.uk/wadler/realworld/`

explicit unbounded storage in the form of a higher-order stack equipped with collapse operation. The computation is modelled by the change of state and operations on the stack. The CPDA model is conceptually easier, at least at first sight. It is used in the fundamental result of Parys cited above. Induction on the stack size of CPDA has been successfully employed in numerous instances. For example, the only approaches known at present to prove the very useful reflection and selection theorems go through CPDA (Broadbent *et al.* 2010; Carayol and Serre 2012). In contrast, λY-calculus has much more structure: it allows a semantic approach using models of the calculus, and it can profit from the rich theory of β-reduction. For example, in this paper we heavily rely on Krivine machine for organizing the reduction process.

The first contribution of this paper is to clarify the correspondence between λY-terms and recursive schemes. Recursive schemes can be seen as a particular form of λY-terms, and indeed the translation from schemes to λY-terms is straightforward. A simple translation in the other direction also exists but makes the type order of the obtained scheme grow unnecessarily. In other words, composing the two translations produces a scheme whose type has order higher than the initial one. Since the order of a scheme is a crucial parameter, this was probably one of the reasons why the community has concentrated its attention on schemes rather than on the λY-calculus (as for example in Broadbent *et al.* (2012)). In this paper, we explain the sources of this mismatch: we introduce a notion of λY-terms in *canonical forms*, show how to translate λY-terms into this form, and prove that the blowup does not happen in this case. This result shows that working in λY-calculus does not induce any handicap as compared to schemes.

The second, and main, result of the paper are two translations of λY-terms to CPDA. They both use Krivine machines to organize reduction of λY-terms. The first works with all λY-terms, and is made optimal with respect to order for terms in canonical forms. Hence, in particular it is also optimal for recursive schemes. The translation shows how to implement the search of the value of a parameter with the help of the *collapse* operation. Indeed the only non-trivial case of the translation is that of variable look-up. The second translation starts immediately from λY-terms in a canonical form, and is also optimal with respect to order. It is more direct, in the sense that it starts with a fixed encoding of environments on the stack, and makes the structure of the stack always reflect this encoding.

The third contribution of the paper is to explain the role of safety in this context. Recall that safe recursive schemes can be translated to CPDA that do not use the *collapse* operation. We introduce a notion of safe λY-terms, and show that our translations applied to safe λY-terms produce automata where the *collapse* operation can be replaced by the *pop* operation. Naturally, the class of safe λY-terms contains all the terms resulting from the translation of safe recursive schemes.

1.1. *Related work*

Until now, most of the related works have used the formalism of recursive schemes rather than λY-calculus. The first translations from schemes to CPDA have been limited

to schemes of order 1 (Aehlig *et al.* 2005; Knapik *et al.* 2005). They used directly the definition of a tree generated by the scheme. They have been based on ideas from Kfoury and Urzyczyn (1988) where a similar translation but for call-by-value calculus has been presented. At that time, this direct approach seemed too cumbersome to generalize to higher orders. The first translation for schemes of all orders (Hague *et al.* 2008) used traversals, a concept based on game semantics (Hyland and Ong 2000). Very recently, Carayol and Serre (2012) have presented a translation extending the one from Knapik *et al.* (2005) to all orders. This translation has been obtained independently from the one presented here. The translation of Carayol and Serre (2012) is limited to recursive schemes, and is based on a mechanism for controlling evaluation specifically invented for the translation. We think that our translations using λY-terms and Krivine machine for evaluation represent a further substantial simplification.

We should also mention the status of the inverse translation: from CPDA to schemes. The translation for order 2 CPDA has been given in Knapik *et al.* (2005) and Aehlig *et al.* (2005), and the general case in Hague *et al.* (2008). While the translation is technical, it represents no fundamental difficulty. The idea is to simulate states of CPDA with some sort of continuations.

The notion of safety has been implicitly introduced in Damm (1982), but had to wait until (Knapik *et al.* 2002) to attract a well-justified attention. Blum and Ong (2009) transfer this notion to simply typed λ-calculus without fixpoints but also to PCF, and study its basic properties. Here we follow this path and adapt the notion to λY-calculus so that it coincides with that of safety as defined for schemes.

The present article is a journal version of Salvati and Walukiewicz (2012).

1.2. *Organization of the paper*

In the next section, we introduce the objects of our study: λY-calculus, recursive schemes, Krivine machine, and CPDA. Section 3 presents translations between recursive schemes and λY-terms. While the translation from schemes to λY-terms is straightforward, the opposite is less so. This leads to a useful notion of λY-terms in a canonical form. Syntactically, terms in the canonical form are in a mid-way between λY-terms and recursive schemes. The two consecutive sections give the two translations from λY-terms to CPDA. The case of safe λY-terms is also analysed there.

2. Basic notions

In this preliminary section, we introduce the basic objects of interest. We start with λY-calculus: a simply-typed λ-calculus with a fixpoint combinator. We then recall the standard notion of the Böhm tree of a λY-term. This permits us to briefly introduce recursive schemes. Later in this section, we present Krivine machines that allow for a more operational way of generating Böhm trees of terms. Finally, we present CPDA and the trees they generate.

2.1. *Simply typed λ-calculus and recursive schemes*

Here we present a classical notion of simply-typed λ-calculus with fixpoints: the *λY-calculus*. We will look at a λY-term as means of generating a potentially infinite tree: the Böhm tree of the term (Barendregt 1977). At the end of the subsection, we present recursive program schemes as λY-terms in a restricted form. In this subsection, the only less standard definitions are that of a tree signature and of the complexity of a term (Definitions 1 and 2). The first is an essential assumption present in all the literature on recursive schemes. The second introduces a measure on terms that we will use frequently.

The *set of simple types* $\mathcal{T}$ is constructed from a unique *basic type* 0 using a binary operation $\rightarrow$. Thus 0 is a type and if α, β are types, so is $(\alpha \rightarrow \beta)$. As usual, so as to use fewer parentheses, we consider that $\rightarrow$ associates to the right. For example, $0 \rightarrow 0 \rightarrow 0$ stands for $0 \rightarrow (0 \rightarrow 0)$. We will write $0^i \rightarrow 0$ as short notation for $0 \rightarrow 0 \rightarrow \cdots \rightarrow 0 \rightarrow 0$, where there are $i+1$ occurrences of 0. The order of a type is defined by: $order(0) = 0$, and $order(\alpha \rightarrow \beta) = max(1 + order(\alpha), order(\beta))$. For a type $\alpha = \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow 0$, we say that α has arity n, which we may write $arity(\alpha) = n$.

A *signature*, denoted Σ, is a set of typed constants, that is symbols with associated types from $\mathcal{T}$. Of particular interest to us will be constants of types of order 1. Observe that types of order 1 have the form $0^i \rightarrow 0$ for some i. The arity of a constant is the arity of its type and we may denote the arity of a constant c with $arity(c)$.

Definition 1. *Tree signature* is a signature where all constants have order at most 1.

In the sequel of the paper, we shall only consider tree signatures, so we always assume that constants have a type with order at most 1.

The set of *simply-typed λY-terms* is defined inductively as follows. A constant of type α is a term of type α. For each type α there is an infinite countable set of variables $x^\alpha, y^\alpha, \ldots$ that are also terms of type α. If M is a term of type β and x^α a variable of type α then $\lambda x^\alpha.M$ is a term of type $\alpha \rightarrow \beta$. If M is a term of type α and x^α is a variable of type α, then $Yx^\alpha.M$ is a term of type α. Finally, if M is of type $\alpha \rightarrow \beta$ and N is a term of type α then (MN) is a term of type β. The order of a term $order(M)$ is the order of its type. In the sequel, we often omit the typing decoration of variables and the unnecessary parentheses following the usual conventions. For technical convenience we do not use Y as a term *per se*, but as a variable binder. This slight modification of the syntax will not affect the computational power of the λY-calculus. The notion of *free variable* is defined as it is usual, and we write $FV(M)$ for the set of free variables of the term M. A term M is *closed* when it has no free variables, i.e. when $FV(M) = \varnothing$. We write $M[N/x]$ for the *capture-avoiding substitution of N for the free occurrences of x in M*. We may write $M[N_1/x_1, \ldots, N_p/x_p]$ for the *simultaneous* capture-avoiding substitutions of N_i for the free occurrences of the x_i in M.

The usual operational semantics of the λ-calculus is given by *β-contraction* ($\rightarrow_\beta$). To give the meaning to the Y-binder we use *δ-contraction* ($\rightarrow_\delta$). These are defined by the rewriting rules:

$$(\lambda x.M)N \rightarrow_\beta M[N/x] \qquad Yx.M \rightarrow_\delta M[Yx.M/x] \ .$$

We write $\rightarrow^*_{\beta\delta}$ for the reflexive and transitive closure of the sum of the two relations. It is called *$\beta\delta$-reduction*. This relation defines an operational equality on terms. We write $=_{\beta\delta}$ for *$\beta\delta$-conversion*, the smallest equivalence relation containing $\rightarrow^*_{\beta\delta}$.

We now define the central notion of *complexity* of λ-terms. This is the principal parameter bounding the length of reductions in terms. As we will see later, this parameter translates into an order of the stack of the simulating CPDA.

Definition 2. Given a term M we define $sub(M)$ to be *the set subterms of M*. It is defined inductively as: $sub(\lambda x.M) = \{\lambda x.M\} \cup sub(M)$; $sub(Yx.M) = \{Yx.M\} \cup sub(M)$, $sub(MN) = \{MN\} \cup sub(M) \cup sub(N)$, $sub(x) = \{x\}$ and $sub(c) = \{c\}$.

The *complexity of a term M*, denoted $comp(M)$, is the maximal order of the type of a subterm of M: $\max\{order(N) \mid N \in sub(M)\}$.

The *variable complexity of a term M*, denoted $vcomp(M)$, is the maximal order of a variable occurring in M (it is 0 if M contains no variables).

For example, consider the term $M = Yx^0.\, ax^0x^0$. The set of its subterms is $sub(M) = \{M, ax^0x^0, ax^0, a, x^0\}$, moreover $comp(M) = 1$ and $vcomp(M) = 0$. As another example take $M' = Yx^{0\rightarrow 0}.\, \lambda y^0.\, y^0$. We have $comp(M') = 1$, as Y is not a subterm of M'. At the same time $vcomp(M') = 1$ too because of the variable x.

Having two measures of complexity allows us to give precise bounds in our results. Fortunately, in most contexts of this paper the two are the same, so they can be confused without much harm. Indeed, when M of type 0 is in β-normal form and $vcomp(M) > 0$ then $vcomp(M) = comp(M)$. So, roughly, *comp* is a good measure for all terms, and *vcomp* is more precise for terms in β-normal form. Notice that, when $vcomp(M) = 0$ we may have $comp(M) = 1$. This is due to occurrences of constants, as in the term $Yx^0.ax^0x^0$.

Lemma 3. Let M be a λY-term of type 0 and in β-normal form. If $vcomp(M) > 0$ then $vcomp(M) = comp(M)$.

Proof. Since every variable occurring in M is a subterm of M we get $vcomp(M) \leqslant comp(M)$. We now prove that every subterm N of M is such that $order(N) \leqslant vcomp(M)$. For this, let us call *rigid*, the subterms of M which have the form $PN_1 \dots N_k$ (k can be 0) where P is either a constant c, a variable x, or a term of the form $Yx.Q$. By induction on the size of a rigid subterm N of M we prove that for all subterms N' of N: $order(N') \leqslant vcomp(M)$. The lemma is a consequence of this property since M is rigid as M is in β-normal form and of type 0.

In case P is a constant, we have $order(N) \leqslant 1$ and as, by assumption, $vcomp(M) > 0$ we have $order(N) \leqslant vcomp(M)$. Since $order(N_i) = 0$ for all $1 \leqslant i \leqslant k$, and since M is in β-normal form, all the N_i's must be rigid. By induction hypothesis we obtain that all the subterms N' of N satisfy $order(N') \leqslant vcomp(M)$.

If P is a variable x or a term $Yx.Q$ then $order(N) \leqslant order(x) \leqslant vcomp(M)$. As M is in β-normal form, $N_i = \lambda x_1 \dots x_{k_i}.N_i'$ where N_i' is rigid. By induction hypothesis, we have that all the subterms of N_i' have order smaller than $vcomp(M)$. Since all terms N_i have order smaller than $order(x)$ we can conclude. □

Another usual reduction rule is *η-contraction* ($\rightarrow_\eta$) defined by the rewriting rule:

$$\lambda x.Mx \rightarrow_\eta M \quad \text{when } x \text{ is not in the set } FV(M) \text{ of free variables of } M.$$

Following our naming conventions, *η-reduction* and *η-conversion* are, respectively, the reflexive transitive closure, and the symmetric reflexive transitive closure of η-contraction. We will not use η-contraction in this paper, but we introduce it so as to explain a particular syntactic presentation of simply typed λY-terms. It is customary in simply typed λ-calculus to work with terms in *η-long forms* that have a convenient property of syntactically reflecting the structure of their typing. A term M is in *η-long* form when each of its subterms is either of type 0, or starts with a λ-abstraction, or is applied to an argument in M. It is known that every simply typed term is η-convertible to a term in η-long form.

For example, the term $M = \lambda x^{0\rightarrow(0\rightarrow0)\rightarrow0}\lambda y^0.\ x^{0\rightarrow(0\rightarrow0)\rightarrow0}y^0$ is not in η-long form. Indeed the subterm $x^{\alpha\rightarrow(\alpha\rightarrow\beta)\rightarrow\beta}y^\alpha$ has type $(0\rightarrow0)\rightarrow0$, but it is not a λ-abstraction, and it is not applied to an argument in M. Its η-long form is

$$M' = \lambda x^{0\rightarrow(0\rightarrow0)\rightarrow0}\lambda y^0\lambda z^{0\rightarrow0}.\ x^{0\rightarrow(0\rightarrow0)\rightarrow0}y^0(\lambda u^0.z^{0\rightarrow0}u^0).$$

We can remark that the subterm $x^{\alpha\rightarrow(\alpha\rightarrow\beta)\rightarrow\beta}y^\alpha$ receives $(\lambda u^0.z^{0\rightarrow0}u^0)$ as an argument in M' and that every subterm of M' satisfies the required properties for M' to be in η-long form.

The operational semantics of the λY-calculus is the $\beta\delta$-reduction. It is well known that $\beta\delta$-reduction is confluent and enjoys subject reduction (i.e. the type of terms is invariant under computation). So every term has at most one normal form, but due to δ-reduction there are terms without a normal form. It is classical in the theory of λ-calculus to consider a kind of infinite normal form that by itself is an infinite tree, and in consequence is not a term of λY-calculus (Barendregt 1977; Barendregt and Klop 2009; Dezani-Ciancaglini *et al.* 1998). We define it below.

A *Böhm tree* is an unranked, ordered, and potentially infinite tree with nodes labelled by terms of the form $\lambda x_1.\ldots x_n.N$ (we may write such a label $\lambda\overrightarrow{x}.N$) where N is a variable or a constant (the sequence of λ-abstractions can be empty); or the symbol ω^α that denotes the undefined Böhm tree of type α. So for example x^0, ω^0 are labels, but $\lambda y^0.x^{0\rightarrow0}\ y^0$ and $\lambda x.\omega^0$ are not.

Definition 4. A *Böhm tree* of a term M is obtained in the following way.

— If $M \rightarrow^*_{\beta\delta} \lambda\overrightarrow{x}.N_0N_1\ldots N_k$ with N_0 a variable or a constant then $BT(M)$ is a tree having root labelled by $\lambda\overrightarrow{x}.N_0$ and having $BT(N_1),\ldots,BT(N_k)$ as subtrees.
— Otherwise $BT(M) = \omega^\alpha$, where α is the type of M.

Notice that the confluence of $\rightarrow^*_{\beta\delta}$, known as the *Church–Rosser property*, guaranties that for every term M, $BT(M)$ is uniquely defined.

Observe that a term M has a $\beta\delta$-normal form if and only if $BT(M)$ is a finite tree with no occurrence of ω. In this case the Böhm tree is just another representation of the normal form. Unlike in the standard theory of the λ-calculus we will be rather interested in terms with infinite Böhm trees.

Recall that in a tree signature all constants are of order at most 1. A closed term in normal form of type 0 over such a signature is just a finite tree, where constants of type 0 are in leaves and constants of a type $0^k \to 0$ are labels of inner nodes with k children. The same holds for Böhm trees:

Lemma 5. If M is a closed term of type 0 over a tree signature then $BT(M)$ is a potentially infinite tree whose leaves are labelled with constants of type 0 (possibly ω^0) and whose internal nodes with k children are labelled with constants of type $0^k \to 0$.

2.1.1. *Higher-order recursive schemes.* use a somehow simpler syntax: the fixpoint operators are implicit and so is the λ-abstraction. A recursive scheme over a finite set of non-terminals $\mathcal{N}$ is a collection of equations, one for each nonterminal. A nonterminal is a typed functional symbol. On the left side of an equation we have a nonterminal, and on the right side a term that is its meaning. For a formal definition, we will need the notion of an *applicative term*, that is a term constructed from non-terminals λ-variables and constants using the application construction. More precisely, fixing a finite set of non-terminals $\mathcal{N}$, the set of applicative terms over $\mathcal{N}$, is the smallest set of λ-terms containing all the nonterminals of $\mathcal{N}$, all λ-variables and all constants; if M and N are applicative terms respectively of type $\alpha \to \beta$ and β then (MN) is in an applicative term of β. Let us fix a tree signature Σ, and a finite set of typed non-terminals $\mathcal{N}$. A higher-order recursive scheme is a function $\mathcal{R}$ assigning to every non-terminal $F \in \mathcal{N}$, a term $\lambda\vec{x}.M_F$ where: (i) M_F is an applicative term over $\mathcal{N}$, (ii) the type of $\lambda\vec{x}.M_F$ is the same as the type of F, and (iii) the free variables of M_F are among $\vec{x}$. Sometimes we will say that right-hand sides of a scheme are applicative terms, although more precisely they are applicative terms preceded by a sequence of λ-abstractions.

Definition 6. The *order of a scheme* $\mathcal{R}$ is the maximal order of the type of its nonterminals. We write $order(\mathcal{R})$ for the order of $\mathcal{R}$.

For example, the following is a scheme of the map function that applies its first argument f to every element of the list l given as its second argument.

$$map^{(0\to0)\to0\to0} \equiv \lambda f^{0\to0}.\lambda l^0.\ \mathbf{if}\,(\mathbf{eqnil}\,l)\,\mathbf{nil}\,(\mathbf{cons}(f\,(\mathbf{head}\,l)))\,(map\,f\,(\mathbf{tail}\,l)) \tag{1}$$

It is a scheme of order 1 which can be seen as the syntactic abstraction of the ML program for the map function where the *if* ... *then* ... *else* ... construct is abstracted with the syntactic ternary tree constructor **if** and the operations on lists are abstracted with the syntactic tree constructors **cons**, **nil**, **head**, **tail** and **eqnil**. These constructors respectively represent the operations that push an element in front of a list; the empty list; the extraction of the top-most element of a list; the operation that removes the top-most element of a list; and the test of whether a list is empty or not.

We will not recall formally here a rather lengthy definition of a tree generated by a recursive scheme referring the reader to Knapik *et al.* (2002) and Damm (1982). But so as to give an overall idea to the reader, a scheme defines a rewriting system over applicative terms; indeed, given a non-terminal F of type $\alpha_1 \to \cdots \to \alpha_n \to 0$ to which the scheme assigns the term $\lambda x_1 \dots x_n.M_F$, if $FN_1 \dots N_n$ occurs in an applicative term M,

then M may be rewritten in M' obtained by replacing that occurrence of $FN_1 \dots N_n$ in M by $M_F[N_1/x_1, \dots, M_n/x_n]$. Applying those rewriting from the starting symbol of the scheme we obtain in the limit the infinite tree generated by the scheme. This definition is sound thanks to the confluence of the rewriting relations induced by schemes. As we shall remark, this infinite tree can simply be seen as the Böhm tree of a term obtained from the translation presented in Section 3.1.

2.2. *Krivine machines*

A Krivine machine (Krivine 2007), is an abstract machine that computes the weak head normal form of a λ-term. For this it uses explicit substitutions, called *environments*. Environments are functions assigning *closures* to variables, and closures themselves are pairs consisting of a term and an environment. This mutually recursive definition is schematically represented by the grammar:

$$C ::= (M, \rho) \qquad \rho ::= \varnothing \mid \rho[x \mapsto C] \ .$$

As in this grammar, we will use $\varnothing$ for the empty environment. The notation $\rho[x \mapsto C]$ represents the environment which associates the same closure as ρ to variables except for the variable x that it maps to C. We require that in a closure (M, ρ), the environment is defined for every free variable of M. Intuitively such a closure denotes a closed λ-term: it is obtained by substituting for every free variable x of M the λ-term denoted by the closure $\rho(x)$. When $\rho(x) = C$, we say that ρ *binds* C *to* x. Given a closure (M, ρ), we say that it has type α when M has type α.

A configuration of the Krivine machine is a triple (M, ρ, S), where M is a term, ρ is an *environment*, and S is a *stack*. A stack is a sequence of closures. By convention the topmost element of the stack is on the left. The empty stack is denoted by ε. The rules of the Krivine machine are as follows:

$$\begin{aligned}
(\lambda x.M, \rho, (N, \rho')S) &\to (M, \rho[x \mapsto (N, \rho')], S) \\
(MN, \rho, S) &\to (M, \rho, (N, \rho)S) \\
(Yx.M, \rho, S) &\to (M, \rho[x \mapsto (Yx.M, \rho)], S) \\
(x, \rho, S) &\to (M, \rho', S) \qquad \text{when } \rho(x) \text{ is defined and equal to } (M, \rho').
\end{aligned}$$

Note that the machine is deterministic. We will write $(M, \rho, S) \to^* (M', \rho', S')$ to say that the Krivine machine goes in some finite number of steps from configuration (M, ρ, S) to (M', ρ', S').

Intuitions behind the rules are rather straightforward. The first rule says that in order to evaluate an abstraction $\lambda x.M$, we should look for the argument at the top of the stack, then we bind this argument to x, and calculate the value of M. To evaluate an application MN we create a closure out of N and the current environment so as to be able to evaluate N correctly when necessary and put that closure on the stack; then we continue to evaluate M. The rule for $Yx.M$ simply amounts to bind the variable x in the environment to the current closure of $Yx.M$ and to calculate M. Finally, the rule for

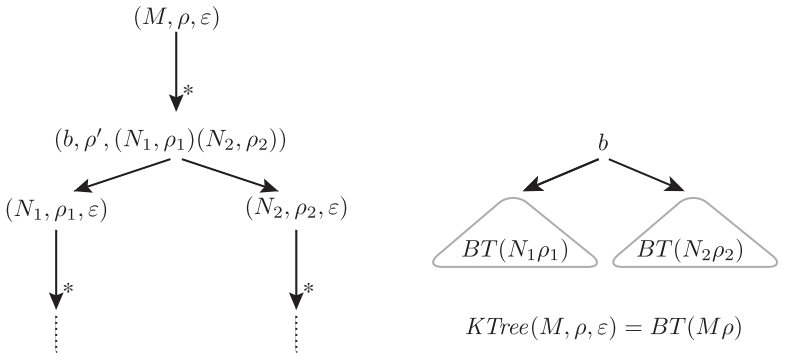


Fig. 1. Computation of Krivine machine and the resulting $KTree(M, \rho, \varepsilon)$.

variables says that we should take the value of the variable from the environment and should evaluate it; the value is not just a term but a closure: a term with an environment giving the right meanings to the free variables of the term.

We will be only interested in configurations accessible from $(M, \varnothing, \varepsilon)$ for some closed term M of type 0. Every such configuration (N, ρ, S) enjoys very strong typing invariants summarized in the following definition and lemma.

Definition 7. Given M a term of type 0, an environment ρ is *M-correct* when for every variable x^α, if $\rho(x^\alpha)$ is defined, then $\rho(x^\alpha)$ is a closure (N, ρ') of type α that is *M-correct*, meaning that:

1. N is in $sub(M)$,
2. ρ' is M-correct, and
3. for every variable y, if $y \in FV(N)$, then $\rho'(y)$ is defined.

A configuration of a Krivine machine (N, ρ, S) is *M-correct* when:

1. (N, ρ) is an M-correct closure, and
2. if N has type $\alpha_n \to \cdots \to \alpha_1 \to 0$, then $S = C_n \dots C_1$ and the closures C_i are M-correct and have types α_i.

Lemma 8. If M is a simply typed term of type 0, given two configurations (N_1, ρ_1, S_1) and (N_2, ρ_2, S_2) so that $(N_1, \rho_1, S_1) \to (N_2, \rho_2, S_2)$, if (N_1, ρ_1, S_1) is M-correct, then (N_2, ρ_2, S_2) is also M-correct.

Let us explain how to use Krivine machines to calculate the Böhm tree of a term (cf. Figure 1). For this we define an auxiliary notion of a tree constructed from a configuration (M, ρ, ε) where M is a term of type 0 over a tree signature. (Observe that the stack should be empty when M is of type 0.) We let $KTree(M, \rho, \varepsilon)$ be the tree consisting only of a root labelled with ω^0 if the computation of the Krivine machine from (M, ρ, ε) does not terminate. If it terminates then $(M, \rho, \varepsilon) \to^* (b, \rho', (N_1, \rho_1) \dots (N_k, \rho_k))$, for some constant b. In this situation $KTree(M, \rho, \varepsilon)$ has b in the root and for every $i = 1, \dots, k$ it has a subtree $KTree(N_i, \rho_i, \varepsilon)$. Due to typing invariants and since we are working with a tree signature, we have that the constant b must have type $0^k \to 0$. In consequence, all terms N_i have type 0.

Definition 9. For a closed term M of type 0, we let $KTree(M)$ be $KTree(M, \varnothing, \varepsilon)$ where $\varnothing$ is the empty environment, and ε is the empty stack.

The next lemma says what $KTree(M)$ is. The proof is immediate from the fact that Krivine machine performs head reduction.

Lemma 10. For every closed term M of type 0 over a tree signature: $KTree(M) = BT(M)$.

Example 1. We give a very simple illustration of the computation of a Böhm tree by the Krivine machine. We compute the Böhm tree of the term M defined as follows:

$$M = \big(\lambda y.\ (\lambda g x.\ Y f.\ g x) a y\big)\ e$$

For readability we adopt the following shorthands:

$$M = (\lambda y.N)e, \quad N = Pay, \quad P = \lambda g x.Y f.g x.$$

We take this example because it illustrates all the reduction rules of the Krivine machine. It also gives us the opportunity to introduce a graphical representation of configurations of the Krivine machine. The reduction sequence is presented in Figure 2 (as expected, the order of execution is represented from left to right and top to bottom).

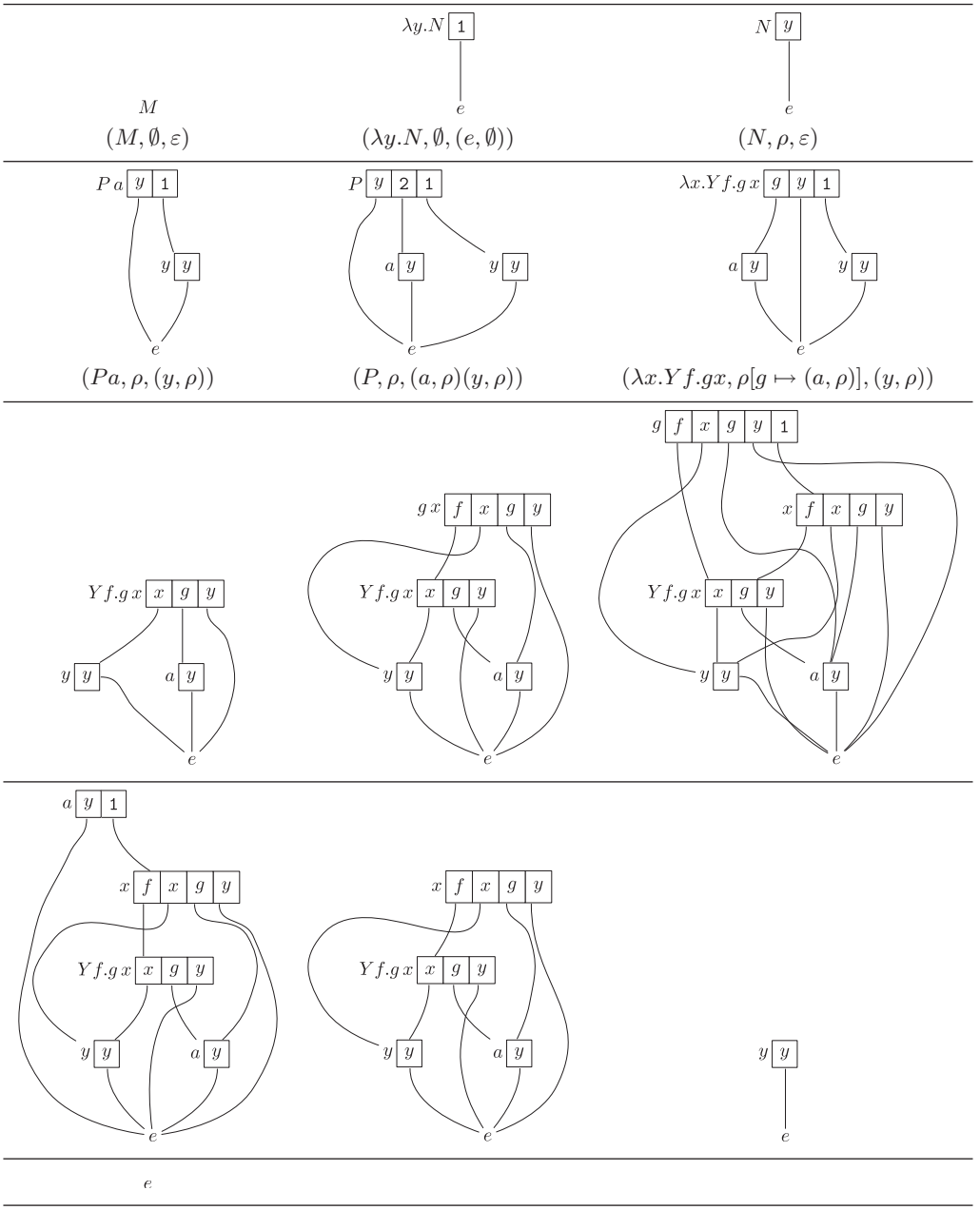
In the pictures of Figure 2, a closure (Q, ρ) is represented by a node labelled by Q followed to the right by boxes containing the variables bound in ρ. Each variable-box is linked to the closure it is bound to. This closure is drawn lower in the graph. When there are no such box on the right, it means that the environment is empty. Finally a configuration $(Q, \rho, C_n \dots C_1)$ is represented similarly to a closure, it is represented by a node labelled Q followed by variable boxes but also by numbered boxes, the box numbered i is linked to the representation of the closure C_i. So as to make the representation sufficiently compact to fit in the paper, we allowed ourselves to make different variables point to the same closure. This graphical sharing is just an artifact of the presentation and even though implementations of the Krivine machines perform some sharing, it is not in general maximal as in our representation.

To make it clear how to interpret those graphical representations as configurations, we have written the six first configurations below their representations.

The ten first configurations correspond to the computation of the label of the root of the Böhm tree. Indeed the tenth configuration is of the form $(a, \rho, (x, \rho'))$. This tells us that the root of $KTree(M, \varnothing, \varepsilon)$ is labelled by a. The eleventh configuration, (x, ρ', ε), starts the computation of the daughter of the root that is simply given by the thirteenth configuration and is e. The Böhm tree of M is therefore $\overset{a}{\underset{|}{e}}$.

2.3. *Collapsible pushdown automata*

CPDA, are like standard pushdown automata, except that they work with a higher-order stack, and can do a *collapse* operation. We will first introduce higher-order stacks and operations on them. Then we will define CPDA, and explain how they can be used to generate infinite trees. In this subsection, we fix a tree signature Σ.



$M = (\lambda y.N)e$, $N = Pay$, $P = \lambda gx.Yf.gx$ and
ρ is the environment such that $\rho(y) = (e, \emptyset)$.

Fig. 2. Computation of the Krivine machine from the configuration $(M, \varnothing, \varepsilon)$.

A stack of order m is a stack of stacks of order $m-1$. Let Γ be a stack alphabet. *An order 0 stack* is a symbol from Γ. *An order m stack* is a non-empty sequence $[S_1 \dots S_l]$ of *order* $(m-1)$ *stacks*. A *higher-order stack* is a stack of order m for some m. The topmost element of an m-stack S, $top(S)$, is S if $m = 0$, and the topmost element of the topmost $(m-1)$-stack of S otherwise. We write $Ord(S) = m$ to denote the fact that S is an m-stack.

2.3.1. *Collapsible pushdown automaton of order m.* (m-CPDA) works with m-stacks. Symbols on stacks are symbols from Γ with a superscript that is a pair of numbers, written $a^{i,k}$. As we will see below, this superscript is a recipe for the *collapse* operation: it means to do k-times the operation pop_i. So k may be arbitrary large but $i \in \{1, \dots, m\}$. We call such a superscript a *pointer of order i*. By abuse of notation $top(S)$ will, most of the time (apart in the definition of the *collapse* operation), refer to a rather than to $a^{i,k}$.

The operations on a stack of order m are indexed by their order $i \in \{1, \dots, m\}$ when needed. We have pop_i, $copy_i$, $push^{a,i}$ for $a \in \Sigma$, and *collapse*. On a stack $S = [S_1 \dots S_{l+1}]$ of order $j \geqslant i$ these operations are defined as follows:

$$pop_i(S) = \begin{cases} [S_1 \dots S_l] & \text{if } i = Ord(S) \text{ and} \\ [S_1 \dots S_l \ pop_i(S_{l+1})] & \text{otherwise,} \end{cases}$$

$$copy_i(S) = \begin{cases} [S_1 \dots S_l S_{l+1} \ S_{l+1}^{\uparrow i}] & \text{if } i = Ord(S) \\ [S_1 \dots S_l \ copy_i(S_{l+1})] & \text{if } i < Ord(S). \end{cases}$$

Here $S_l^{\uparrow i}$ is S_l with all the superscripts (i, k_i), for some k_i, replaced by $(i, k_i + 1)$. Usually the operation $copy_1$ is not allowed in the definition of CPDA. Here we allow it, but it does not change the expressive power of CPDA.

$$push^{a,i}(S) = \begin{cases} [S_1 \dots S_l S_{l+1} a^{i,1}] & \text{if } Ord(S) = 1 \\ [S_1 \dots S_l \ push^{a,i}(S_{l+1})] & \text{otherwise.} \end{cases}$$

So we can push new elements only on the topmost order 1 stack: the *copy* operation allows one to replicate a stack of a given order and is thus responsible for the creation of stacks of order strictly greater than 1. Observe that with a push operation, we also specify the order of the pointer that is attached to the letter we put on the stack.

$$collapse(S) = pop_i^k(S) \quad \text{where } top(S) = a^{i,k}, \text{ for some } a \in \Gamma.$$

We write $pop_i^k(S)$ as a shorthand for applying k times the operation pop_i to S. Thus, the collapse operation performs the pop_i operation k times, where k and i are specified by the pointer, i.e. superscripts attached to the topmost letter of the stack.

A CPDA of order m is a tuple $\mathcal{A} = \langle \Sigma, \Gamma, Q, q_0, \delta \rangle$, where Σ is the tree signature, Γ is the stack alphabet, Q is a finite set of states, q_0 is an initial state, and δ is a transition function:

$$\delta : Q \times \Gamma \to (Q \times Op^m(\Gamma)) \cup \bigcup_{b \in \Sigma} (\{b\} \times Q^{arity(b)}),$$

where $Op^m(\Gamma)$ is the set of stack operations of order m obtained by composing pop_i, $copy_i$, $push^{a,i}$ and *collapse* with $1 \leqslant i \leqslant m$ and $a \in \Gamma$.

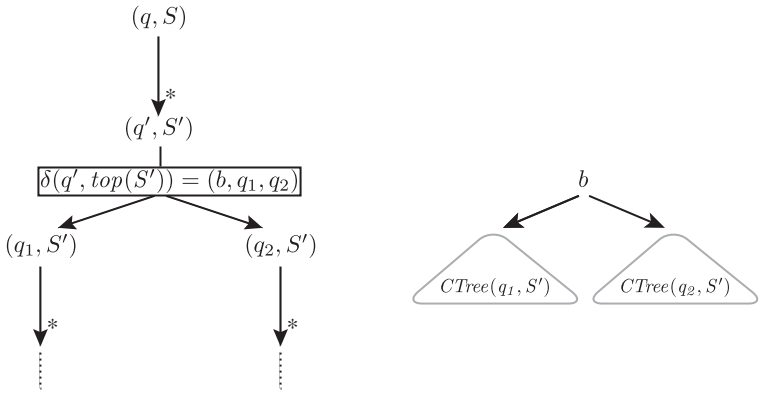


Fig. 3. Computation of CPDA and the resulting $CTree(M, \rho, \varepsilon)$.

The idea is that a state and a top stack symbol (without its superscript pair of numbers) determine either a stack operation or a constant that the automaton is going to produce. The arity of the constant determines the number of new branches of the computation of the automaton. As usual, we will suppose that there is a symbol $\bot \in \Gamma$ used to denote the bottom of the stack. We will also denote by $\bot$ the initial stack containing only $\bot$.

We now explain how a CPDA $\mathcal{A}$ produces a tree when started in a state q with a stack S. We let $CTree(q, S)$ to be a tree consisting of only a root labelled ω^0 if from (q, S) the automaton does an infinite sequence of stack operations. Otherwise from (q, S) after a finite number of stack operations the automaton arrives at a configuration (q', S') with $\delta(q', top(S')) = (b, q_1, \ldots, q_k)$ for some constant b. In this situation $CTree(q, S)$ has the root b and for every $i = 1, \ldots, k$ it has $CTree(q_i, S')$ as a subtree (Figure 3).

Definition 11. For a CPDA $\mathcal{A}$ we let $CTree(\mathcal{A})$ be $CTree(q^0, \bot)$; where q^0 is the initial state of $\mathcal{A}$, and $\bot$ is the initial stack.

3. From recursive schemes to λY-calculus and back

We present a translation of recursive schemes to λY-terms such that the tree generated by a scheme is a Böhm tree of a term obtained from the translation (Lemma 12). We also show how λY-terms can be translated into schemes (Theorem 19). The latter translation is an inverse of the first one in a sense that composed together the two translations do not increase the order of the scheme. To obtain this property we need to have a closer look at the structure of λY-terms, and introduce a notion of canonical form. This form will be also very useful in the context of CPDA in later sections.

3.1. *From recursive schemes to λY-calculus*

The translation from a recursive scheme to a λ-term is given by a standard variable elimination procedure, using the fixpoint binder Y. Suppose $\mathcal{R}$ is a recursive scheme over a set of non-terminals $\mathcal{N} = \{F_1, \ldots, F_n\}$. The term T_n representing the meaning of the

non-terminal F_n is obtained as follows:

$$\begin{aligned} T_1 &= Y\,F_1.\mathcal{R}(F_1) \\ T_2 &= Y\,F_2.\mathcal{R}(F_2)[T_1/F_1] \\ &\vdots \\ T_n &= Y\,F_n.(\ldots((\mathcal{R}(F_n)[T_1/F_1])[T_2/F_2])\ldots)[T_{n-1}/F_{n-1}]. \end{aligned} \tag{2}$$

The translation (2) applied to the recursion scheme for *map* (*c.f.* (1) p. 8) gives a term:

$$Y\,map^{(0\to 0)\to 0\to 0}.\lambda f^{0\to 0}.\lambda l^{0}.\,\mathbf{if}\ (\mathbf{eqnil}\,l)\,\mathbf{nil}\,\big(\mathbf{cons}\,(f\,(\mathbf{head}\,l)))\,(map\,f\,(\mathbf{tail}\,l))$$

For completeness, we state the equivalence property. Anticipating contexts where the order of a scheme or a term is important let us observe that the complexity of the term obtained from the translation may be smaller than the order of the scheme.

Lemma 12. Let $\mathcal{R}$ be a recursion scheme and let F_n be one of its nonterminals. A term T_n obtained by the translation (2) is such that $BT(T_n)$ is the tree generated by the scheme from non-terminal F_n. Moreover $vcomp(T_n) \leqslant order(\mathcal{R})$.

The reason why we have $vcomp(T_n) \leqslant order(\mathcal{R})$ and not $vcomp(T_n) = order(\mathcal{R})$ is that in a scheme $\mathcal{R}$ there may be nonterminals that are not accessible from the starting symbol and that the process described in (2) may erase certain non-accessible nonterminals. We use *vcomp* instead of *comp* in the above lemma, but the two are almost the same since T_n is in β-normal form (cf. Lemma 3).

Let us take the scheme defined by the equations:

$$\begin{aligned} S &\equiv Ab \\ A &\equiv \lambda x.ax(Ax) \\ B &\equiv \lambda f.fI \\ I &\equiv \lambda x.x \end{aligned}$$

The λY-term that we obtain by applying the translation (2) is the term

$$M = Y\,S.(Y\,A.\lambda x.ax(Ax))b.$$

While M has variable complexity 1, the scheme has order 2 because of the non-accessible non-terminal B. Of course, for schemes where all nonterminals are accessible, the variable complexity of the λY-term obtained from translation (2) is equal to the order of the scheme.

3.2. *From λY-calculus to recursive schemes*

As already noted by Damm (1982), recursive schemes are a representation of the whole λY-calculus. Yet we think that it is very instructive to spell out a translation. We are actually going to present two translations. The first one will be rather straightforward and will not assume any particular property of λY-terms that it transforms. However it will not be dual to the one from Lemma 12 above in the sense that applying the

two translations one after another can increase the order of a scheme by 1. To obtain a dual translation, we will first need to transform a λY-term into a form that we call *canonical*. As we will see later, the translation from Lemma 12 produces directly a term in a canonical form. We will present a translation of canonical terms into schemes that gives a scheme of the expected order: translating a scheme to a λY-term using Lemma 12 and back to scheme does not increase the order (Theorem 19).

For the first translation, we assume that the bound variables of M are pairwise distinct and that they are totally ordered; thanks to this total order, we associate to each subterm N of M the sequence of its free variables $SV(N)$, that is the set of free variables of N ordered with respect to that total order. We then define a recursive scheme $\mathcal{R}_M$ whose set of nonterminals is $\mathcal{N}_M = \{\langle N\rangle \mid N \in sub(M)\}$; if $\langle N\rangle$ is in $\mathcal{N}_M$, $SV(N) = x_1^{\alpha_1} \dots x_n^{\alpha_n}$ and N has type α, then $\langle N\rangle$ has type $\alpha_1 \to \cdots \to \alpha_n \to \alpha$. Each element $\langle N\rangle$ of $\mathcal{N}_M$ determines a single equation in $\mathcal{R}_M$ as described in the following table:

Term	The associated equation
y	$\langle y\rangle \equiv \lambda y.y$
a	$\langle a\rangle \equiv a$
N_1N_2	$\langle N_1N_2\rangle \equiv \lambda\overrightarrow{x}.\langle N_1\rangle\overrightarrow{y}(\langle N_2\rangle\overrightarrow{z})$ where $SV(N_1N_2) = \overrightarrow{x}$, $SV(N_1) = \overrightarrow{y}$ and $SV(N_2) = \overrightarrow{z}$
$\lambda y.P$	$\langle \lambda y.P\rangle \equiv \lambda\overrightarrow{x}y.\langle P\rangle\overrightarrow{z}$ where $SV(\lambda y.P) = \overrightarrow{x}$ and $SV(P) = \overrightarrow{z}$
$Yx.P$	$\langle Yx.P\rangle \equiv \lambda\overrightarrow{z}.\langle P\rangle\overrightarrow{z_1}(\langle Yx.P\rangle\overrightarrow{z})\overrightarrow{z_2}$ when $x \in FV(P)$, $SV(Yx.P) = \overrightarrow{z}$ and $SV(P) = \overrightarrow{z_1}x\overrightarrow{z_2}$
$Yx.P$	$\langle Yx.P\rangle \equiv \lambda\overrightarrow{z}.\langle P\rangle\overrightarrow{z}$ when $x \notin FV(P)$ and $SV(P) = \overrightarrow{z}$

The next lemma summarizes the properties of the translation. We omit the proof: it follows directly from the definitions.

Lemma 13. For every term M the tree generated from non-terminal $\langle M\rangle$ by $\mathcal{R}_M$ defined in the table above is identical to $BT(M)$. The order of $\mathcal{R}_M$ is at most $comp(M) + 1$.

The order of the obtained scheme does not correspond to the order of the translation from schemes to λY-terms. In principle, each time we compose the two translations, the order of the scheme we obtain is increased by 1 with respect to the original scheme.

The problem comes from the fact that all the variables in a λY-term are treated uniformly while nonterminals, that correspond to variables bound by Y in λY-terms, in schemes have a special treatment.

Example 2. Let us take the λY-term $M = Yx^{0\to 0}.\lambda z^0.a(x^{0\to 0}z^0)e$ where a and e are constants with respective types $0 \to 0$ and 0. The scheme $\mathcal{R}_M$ obtained by translating M

as described above is:

$$\begin{aligned}
\langle M\rangle &\equiv \langle Yx^{0\to 0}.\lambda z^0.a(x^{0\to 0}z^0)\rangle\langle e\rangle\\
\langle e\rangle &\equiv e\\
\langle Yx^{0\to 0}.\lambda z^0.a(x^{0\to 0}z^0)\rangle &\equiv \langle \lambda z^0.a(x^{0\to 0}z^0)\rangle\langle Yx^{0\to 0}.\lambda z^0.a(x^{0\to 0}z^0)\rangle\\
\langle \lambda z^0.a(x^{0\to 0}z^0)\rangle &\equiv \lambda x^{0\to 0}z^0.\langle a(x^{0\to 0}z^0)\rangle x^{0\to 0}z^0\\
\langle a(x^{0\to 0}z^0)\rangle &\equiv \lambda x^{0\to 0}z^0.\langle a\rangle(\langle x^{0\to 0}z^0\rangle x^{0\to 0}z^0)\\
\langle a\rangle &\equiv a\\
\langle x^{0\to 0}z^0\rangle &\equiv \lambda x^{0\to 0}z^0.\langle x^{0\to 0}\rangle x^{0\to 0}(\langle z^0\rangle z^0)\\
\langle x^{0\to 0}\rangle &\equiv \lambda x^{0\to 0}.x^{0\to 0}\\
\langle z^0\rangle &\equiv \lambda z^0.z^0.
\end{aligned}$$

While $comp(M) = 1$, the order of the scheme is 2 because every nonterminal of the form $\langle N\rangle$, for N a subterm of M containing both $x^{0\to 0}$ and z^0, has type $(0 \to 0) \to 0 \to 0$ which is of order 2.

Example 3. If we now take $M = Yx^0.ax^0x^0$, the translation will yield the following scheme:

$$\begin{aligned}
\langle M\rangle &\equiv \langle axx\rangle\langle M\rangle\\
\langle axx\rangle &\equiv \lambda x.\langle ax\rangle x(\langle x\rangle x)\\
\langle ax\rangle &\equiv \lambda x.\langle a\rangle(\langle x\rangle x)\\
\langle a\rangle &\equiv a\\
\langle x\rangle &\equiv \lambda x.x\ .
\end{aligned}$$

The scheme has order 1 since, except $\langle M\rangle$, all nonterminals have order 1. Observe that $comp(M) = 1$ but $vcomp(M) = 0$. So even in that case we do not obtain an order matching that from Lemma 12. This example shows that we should treat in a special way not only recursive variables but also constants.

As a first step towards obtaining a better translation from terms to schemes we need to make the distinction between the variables that are bound by a λ and the ones that are bound by a Y.

Definition 14. For each type α the set of variables of type α is partitioned into two infinite sets: the set of *λ-variables* and the set of *Y-variables* which can respectively only be bound with λ or Y.

We will mark this distinction between λ-variables and Y-variables by writing Y-variables in boldface font.

From now on, we will assume that every Y-variable in a term M is bound at most once. This means that, given a Y-variable $\mathbf{x}$ of M, we can write $term_M(\mathbf{x})$ for the unique subterm $Y\mathbf{x}.N$ of M starting with the binder of $\mathbf{x}$. We will omit a subscript M if it is clear from the context.

Definition 15. The term M is in *canonical form* when it satisfies the following two properties:

1. M is in β-normal form: it has no subterms of the form $(\lambda x.P)Q$.
2. If $Y\,\mathbf{x}.N$ is a subterm of M then all the free variables in N are Y-variables.

It is worth noticing that the translation from schemes to terms we have described in the previous subsection produces terms in canonical form. We first note an interesting property of terms in β-normal form over a tree signatures.

Lemma 16. Let M be a term over a tree signature. If M is closed of type 0 and in β-normal form then for every λ-variable z in M there is a Y-variable $\mathbf{x}$ in M such that $order(z) < order(\mathbf{x})$.

Proof. Since M is closed, the variable z is bound somewhere in M. We proceed by induction on the depth to which z is bound (more precisely, we here mean the distance from the root to the node of the λ-abstraction binding z in the syntactic tree of M). As M is in β-normal form we have two cases:

— the variable z appears in the sequence $\overrightarrow{z}$ in a term $Y\,\mathbf{x}.(\lambda\overrightarrow{z}.Q)$. Here we get immediately that $order(z) < order(\mathbf{x})$ from the fact that the type of $\lambda\overrightarrow{z}.Q$ is the same as the type of $\mathbf{x}$.

— Otherwise z is a variable in one of the sequences $\overrightarrow{z_1}, \ldots, \overrightarrow{z_p}$ in a subterm like $P(\lambda\overrightarrow{z_1}.N_1)\ldots(\lambda\overrightarrow{z_p}.N_p)$ where, either $P = y$, or $P = \mathbf{x}$, or $P = Y\,\mathbf{x}.Q$. Observe that P cannot be a constant, since all our constants are of order at most 1 so the sequences $\overrightarrow{z_1}, \ldots, \overrightarrow{z_p}$ would be empty. In case $P = y$, we have $order(z) < order(y)$. But the λ-variable y is bound at a shorter depth than z and by induction there is a Y-variable $\mathbf{x}$ such that $order(y) < order(\mathbf{x})$ and therefore $order(z) < order(\mathbf{x})$ which gives the desired result. In both the cases where $P = \mathbf{x}$ and where $P = Y\,\mathbf{x}.Q$ we obviously have $order(z) < order(\mathbf{x})$.

□

As expected, every term can be put into a canonical form.

Lemma 17. For every term M over a tree signature and of type 0 there is a term M' in canonical form such that $BT(M') = BT(M)$ and $vcomp(M') \leqslant vcomp(M)$.

Proof. Consider a term M_0 of type 0. The first step is to normalize M_0 with respect to β-reduction without performing δ-reductions. It is well known that simply typed λ-calculus has the strong normalization property so this procedure terminates. Let M_1 be the resulting term. By definition $BT(M_1)$ is the same as $BT(M_0)$ and $vcomp(M_1) \leqslant vcomp(M_0)$. Moreover, M_1 satisfies the first condition of being in a canonical form (cf. Definition 15).

The second step involves removing λ-variables from fixpoint subterms. We replace, starting from outermost subterms, every subterm $Y\,\mathbf{x}^{\alpha}.P$ of M_1 by $Q\,\overrightarrow{y}$ where

$$Q = Y\,\mathbf{z}^{\gamma}.\lambda\overrightarrow{y}.P\,[\mathbf{z}^{\gamma}\,\overrightarrow{y}/\mathbf{x}^{\alpha}],$$

$\overrightarrow{y} = y_1^{\beta_1} \dots y_m^{\beta_m}$ is a sequence of the λ-variables that are free in P, $\gamma = \beta_1 \to \dots \to \beta_m \to \alpha$ and $\mathbf{z}^\gamma$ is a fresh Y-variable. Notice that the replacement may introduce new free variables to some subterms. As $order(\alpha) \leqslant vcomp(M_1)$, and as Lemma 16 implies that the orders of $\beta_1, \dots, \beta_m$ are strictly smaller than $vcomp(M_1)$ we have $order(\gamma) \leqslant vcomp(M_1)$. This transformation thus gives a term M_2 such that $vcomp(M_2) = vcomp(M_1)$. To show that $BT(M_2) = BT(M_1)$ it is sufficient to show that $Q\overrightarrow{y}$ and $Y\mathbf{x}^\alpha.P$ have the same Böhm trees. By the fact that equivalence of Böhm trees is a congruence for λ-calculus (see Barendregt (1985)), we will get the desired identity $BT(M_2) = BT(M_1)$. To obtain $BT(Q\overrightarrow{y}) = BT(Y\mathbf{x}^\alpha.P)$, let us first remark that:

$$\begin{aligned} Q\overrightarrow{y} &= (Y\mathbf{z}^\gamma \lambda\overrightarrow{y}.P[\mathbf{z}^\gamma\overrightarrow{y}/\mathbf{x}^\alpha])\overrightarrow{y} \\ &\to_\delta (\lambda\overrightarrow{y}.P[Q\overrightarrow{y}/\mathbf{x}^\alpha])\overrightarrow{y} \\ &\to_\beta^* P[Q\overrightarrow{y}/\mathbf{x}^\alpha]). \end{aligned}$$

By iterating this sequence of reduction steps we obtain

$$\begin{aligned} Q\overrightarrow{y} &\to_{\delta\beta}^* P[Q\overrightarrow{y}/\mathbf{x}^\alpha]) \\ &\to_{\beta\delta}^* P[P[Q\overrightarrow{y}/\mathbf{x}^\alpha]/\mathbf{x}^\alpha] \\ &\to_{\beta\delta}^* P[P[\dots[Q\overrightarrow{y}/\mathbf{x}^\alpha]\dots]/\mathbf{x}^\alpha]. \end{aligned}$$

This identity and an easy induction shows that $BT(Q\overrightarrow{y}) = BT(Y\mathbf{x}^\alpha.P)$.

Therefore, we have that M_2 satisfies all the conditions of being in a canonical form. Moreover, we have $BT(M_2) = BT(M_0)$ and $vcomp(M_2) \leqslant vcomp(M_0)$. □

It remains to present a translation of terms M in canonical form into schemes $\mathcal{R}_M$ such that $order(\mathcal{R}_M) = vcomp(M)$. This is made possible thanks to the distinction between the two kinds of variables. The scheme obtained by this translation will use λ-abstraction only for λ-variables of the original λY-term and not, as in the previous translation, for the Y-variables.

Again we assume that we have a fixed total order on variables in M. For a subterm N of M, we let $\lambda SV(N)$ be the sequence, ordered according to the fixed total order, of λ-variables that are free in N. The non-terminals $\mathcal{N}_M$ of $\mathcal{R}_M$ are $[N]$ where N is in $isub(M)$ the set of *interesting terms of* M. This set is recursively defined as follows: $isub(\lambda x.M) = \{\lambda x.M\} \cup isub(M)$, $isub(Yx.M) = \{Yx.M\} \cup isub(M)$, $isub(aN_1 \dots N_n) = \{aN_1 \dots N_n\} \cup \bigcup_{i=1}^n isub(N_i)$, $isub(x) = \{x\}$, $isub(c) = \{c\}$; $isub(MN) = \{MN\} \cup isub(M) \cup isub(N)$ (when M is not of the form $aN_1 \dots N_p$ for some constant a). The notion of interesting subterms is very similar to the notion of subterm; the only difference is that when $aN_1 \dots N_k$ is a subterm of M then for every $i < k$, $aN_1 \dots N_i$ is not an interesting subterm of M. This small technical difference is used to avoid the effects of constants as illustrated in Example 3.

If N has type α and $[N]$ is in $\mathcal{N}_M$, then its type will be $\alpha_1 \to \dots \to \alpha_n \to \alpha$ when $\lambda SV(N) = x_1^{\alpha_1} \dots x_n^{\alpha_n}$. We associate with each element $[N]$ of $\mathcal{N}_M$ an equation according to the following table:

Term	The associated equation
y	$[y] \equiv \lambda y.y$ when y is a λ-variable
$\mathbf{x}$	$[\mathbf{x}] \equiv [term(\mathbf{x})]$ when $\mathbf{x}$ is a Y-variable
$aN_1 \dots N_p$	$[aN_1 \dots N_n] = \lambda\overrightarrow{x}.a([N_1]\overrightarrow{x_1}) \dots ([N_p]\overrightarrow{x_p})$ where $\lambda SV(aN_1 \dots N_p) = \overrightarrow{x}$, $\lambda SV(N_i) = \overrightarrow{x_i}$ for $i \in [1,p]$.
N_1N_2	$[N_1N_2] \equiv \lambda\overrightarrow{x}.[N_1]\overrightarrow{y}([N_2]\overrightarrow{z})$ where $\lambda SV(N_1N_2) = \overrightarrow{x}$, $\lambda SV(N_1) = \overrightarrow{y}$, and $\lambda SV(N_2) = \overrightarrow{z}$
$\lambda y.P$	$[\lambda y.P] \equiv \lambda\overrightarrow{x}y.[P]\overrightarrow{z}$ where y is a λ-variable, $\lambda SV(\lambda y.P) = \overrightarrow{x}$, and $\lambda SV(P) = \overrightarrow{z}$
$Y\,\mathbf{x}.P$	$[Y\,\mathbf{x}.P] \equiv [P]$

Let us compare this translation to the previous one. First, Y-variables are not translated at all. The case for abstraction is as before but notice that the rule for $[Y\,\mathbf{x}.P]$ is particularly simple, this is mostly because Y-variables do not need to be passed as parameters because recursive terms do not contain free λ-variable. There is also a slight modification in the case of application. When the head of a sequence of applications is a constant, we do not decompose the applications in the translation. If we would do this then the order of the obtained scheme would be at least 1 because of the order of constants.

Example 4. Consider the term $M = Y\,\mathbf{x}^{0\to 0}.\lambda z^0.\, a(\mathbf{x}^{0\to 0}z^0)e$ from Example 2 . Notice that M is in canonical form so that we can use the translation we have just given. The scheme $\mathcal{R}_M$ obtained by translating M is then:

$$\begin{aligned}
[M] &\equiv [Y\,\mathbf{x}^{0\to 0}.\lambda z^0 a(\mathbf{x}^{0\to 0}z^0)][e]\\
[e] &\equiv e\\
[Y\,\mathbf{x}^{0\to 0}.\lambda z^0 a(\mathbf{x}^{0\to 0}z^0)] &\equiv [\lambda z^0.a(\mathbf{x}^{0\to 0}z^0)]\\
[\lambda z^0.a(\mathbf{x}^{0\to 0}z^0)] &\equiv \lambda z^0.[a(\mathbf{x}^{0\to 0}z^0)]z^0\\
[a(\mathbf{x}^{0\to 0}z^0)] &\equiv \lambda z^0.a([\mathbf{x}^{0\to 0}z^0]z^0)\\
[\mathbf{x}^{0\to 0}z^0] &\equiv \lambda z^0.[\mathbf{x}^{0\to 0}]([z^0]z^0)\\
[\mathbf{x}^{0\to 0}] &\equiv [Y\,\mathbf{x}^{0\to 0}.\lambda z^0 a(\mathbf{x}^{0\to 0}z^0)]\\
[z^0] &\equiv \lambda z^0.z^0.
\end{aligned}$$

Here the maximal order of a nonterminal of $\mathcal{R}_M$ is 1 while $vcomp(M) = comp(M) = 1$.

Example 5. If we take the term $M = Y\,\mathbf{x}^0.\, a\mathbf{x}^0\mathbf{x}^0$ from Example 3, now we would get the scheme:

$$\begin{aligned}
[M] &\equiv [a\mathbf{x}^0\mathbf{x}^0]\\
[a\mathbf{x}^0\mathbf{x}^0] &\equiv a[\mathbf{x}^0][\mathbf{x}^0]\\
[\mathbf{x}^0] &\equiv [M].
\end{aligned}$$

This scheme is of order 0 as desired since $vcomp(M) = 0$.

Lemma 18. For every λY-term M of type 0 in canonical form, the scheme $\mathcal{R}_M$ defined by the above table generates the tree $BT(M)$, and moreover $order(\mathcal{R}_M) = vcomp(M)$.

Proof. The proof that the tree generated by $\mathcal{R}_M$ is $BT(M)$ follows from comparing reduction sequences, and we will not present it here. We are going to explain why $order(\mathcal{R}) = vcomp(M)$. Recall that $order(\mathcal{R})$ is the maximal order of a nonterminal in $\mathcal{R}$ and that $vcomp(M)$ is the maximal order of a variable occurring in M. In case M does not contain variables, M is a finite term in normal form of type 0 and is thus a finite tree. Then $vcomp(M) = 0$ and, obviously, the order of $\mathcal{R}_M$ is also 0.

In the case $vcomp(M) = 0$ and M contains occurrences of variables, then from Lemma 16, we have that M does not contain any λ-abstraction. A simple induction on the structure of M shows that $order(\mathcal{R}_M) = 0$.

Let us now assume that $vcomp(M) > 0$. By Lemma 16 it must contain a Y-variable and we know that $vcomp(M) = order(\mathbf{x})$ for some Y-variable $\mathbf{x}$ occurring in M. Moreover, from Lemma 3 we have that for every term N in $sub(M)$, $order(N) \leqslant order(\mathbf{x})$. Now take a nonterminal $[N]$ of $\mathcal{R}_N$, and let α the type of N. By definition $[N]$ has type $\alpha_1 \to \cdots \to \alpha_n \to \alpha$ where $\alpha_1, \ldots, \alpha_n$ are the types of the λ-variables occurring free in N. From Lemma 16, we get $order(\alpha_i) < order(\mathbf{x})$ for all i in $[1, n]$ and, as $order(\alpha) \leqslant order(\mathbf{x})$, we have $order(\alpha_1 \to \cdots \to \alpha_n \to \alpha) \leqslant order(\mathbf{x})$. It then follows that $order([N]) \leqslant vcomp(M)$.

□

By combining Lemmas 18 and 17 we obtain:

Theorem 19. For every closed term M over a tree signature and of type 0 there is a scheme $\mathcal{R}$ generating the tree $BT(M)$ and such that $order(\mathcal{R}) \leqslant vcomp(M)$. In the special case when M is in canonical form and $vcomp(M) = 0$ then $\mathcal{R}$ is of order 0 too.

4. From λY-calculus to CPDA

In this section, we will show how to construct for a λY-term M_0 a CPDA $\mathcal{A}$ such that the tree generated by $\mathcal{A}$, that is $CTree(\mathcal{A})$, is the Böhm tree of M_0, i.e, $BT(M_0)$. Put together with the results from the previous section this will also give us a translation for recursive schemes. The construction uses the characterization of Böhm trees in terms of computations of the Krivine machine. Thanks to this characterization it is enough to simulate the Krivine machine using CPDA.

The first step will be to change the representation of the configurations of the Krivine machine. This is done purely for reasons of exposition. Then we will present the construction of the CPDA $\mathcal{A}$ simulating the behaviour of the Krivine machine on a fixed term M_0. From the correctness of the simulation it will follow that $CTree(\mathcal{A}) = KTree(M_0) = BT(M_0)$ (Theorem 26). The order of the stack of $\mathcal{A}$ can be as high as $comp(M_0) + 1$. Put together with the translation from Lemma 12 this does not give an optimal (with respect to order) translation from recursive schemes to CPDA. In the following subsection, we explain how to avoid this problem using some simple manipulations on λY-terms and Krivine machines. In the last subsection, we introduce the notion of safe λY-terms and show that if the translation is applied to such a term

then all *collapse* operations are actually *pop* operations. In other words, as in Carayol and Serre (2012), the general translation gives directly CPDA that do not use the collapse operation.

For the entire section we fix a tree signature Σ.

4.1. *Stackless Krivine machines*

From Lemma 8, it follows that the initial term M_0 determines a bound on the size of the stack in reachable configurations of a Krivine machine. Hence one can eliminate the stack at the expense of introducing auxiliary variables. This has two advantages: the presentation is more uniform, and there is no confusion between the stack of the Krivine machine and the stack of the CPDA. From now on we shall call *max* the bound on the size of the stack in the configurations reachable from $(M_0, \varnothing, \varepsilon)$.

We will use a variable γ_i to represent the ith element of the stack of the Krivine machine. Technically we will need one variable γ_i for every type, but since this type can be always deduced from the value we will omit it. With the help of these variables we can make the Krivine machine *stackless*. Nevertheless we still need to know how many elements there are on the stack. This, of course, by Lemma 8, can be deduced from the type of M, but we prefer to keep this information explicitly for the sake of clarity. So the configurations of the new machine are of the form (M, ρ, k) where k is the number of arguments M requires, that is the size of the stack that the usual configuration of the Krivine machine would have. The new rules of the Krivine machine become:

$$\begin{aligned}
(\lambda x.M, \rho, k) &\to (M, \rho[x \mapsto \rho(\gamma_k)][\gamma_k \mapsto \bot], k-1)\\
(MN, \rho, k) &\to (M, \rho[\gamma_{k+1} \mapsto (N, \rho)], k+1)\\
(Yx.M, \rho, k) &\to (M, \rho[x \mapsto (Yx.M, \rho)], k)\\
(z, \rho, k) &\to (N, \rho'', k) \qquad \text{where } (N, \rho') = \rho(z)\\
&\qquad\qquad \text{and } \rho'' = \rho'[\gamma_1 \mapsto \rho(\gamma_1), \ldots, \gamma_{max} \mapsto \rho(\gamma_{max})].
\end{aligned}$$

There are two novelties in these rules. The first is in the variable rule where the stack variables of ρ' are overwritten with the values they have in ρ. The second one is in the abstraction rule, where the value of the stack variable is used. Observe that due to the form of the rules, if x is a normal variable and $\rho(x) = (N, \rho')$ then N is a normal term (not a stack variable) and the values associated to stack variables in ρ' are never going to be used in the computation since, as we already mentioned, each time a closure is invoked with the variable rule, the values of the stack variables are overwritten. Let $strip(\rho)$ denote the result of removing, recursively, values of stack variables from ρ. This means that for every normal variable x: $strip(\rho)(x) = (N, strip(\rho'))$ where $\rho(x) = (N, \rho')$.

We say that a configuration (M', ρ', k) *represents* a configuration (M, ρ, S) if

— $M' = M$,
— $\rho = strip(\rho')$,

— k is the number of elements on the stack S, and for $(N_i, \rho_i) = \rho'(\gamma_i)$ we have that $(N_i, strip(\rho_i))$ is the ith element on S for $i = 1, \ldots, k$; with 1 being at the bottom of the stack. Moreover $\rho'(\gamma_i) = \bot$ for $i > k$.

The following simple lemma says that the stackless machine behaves in the same way as the original one.

Lemma 20. Suppose that (M', ρ', k') represents (M, ρ, S). There is a transition from (M', ρ', k') iff there is a transition from (M, ρ, S). Moreover, if $(M', \rho', k') \to (M'_1, \rho'_1, k'_1)$ and $(M, \rho, S) \to (M_1, \rho_1, S_1)$ then (M'_1, ρ'_1, k'_1) represents (M_1, ρ_1, S_1).

Thanks to this lemma that we can use stackless Krivine machines for constructing $KTree(M)$ (cf. Definition 9) which is no other than $BT(M)$. Moreover, we shall use for configurations of the stackless Krivine machine the same graphical representations as the one we used in Example 1. The configuration (M, ρ, k) of a Krivine machine is represented with k boxes numbered from k to 1, where the box numbered i being linked to the representation of the closure associated to γ_i in ρ.

4.2. *Simulation*

Fix a closed term M_0, let $m_0 = comp(M_0) + 1$ and let max be the maximal size of the stack among the configurations of the Krivine machine reachable from $(M_0, \varnothing, \varepsilon)$. We want to simulate the computation of the stackless Krivine machine on M_0 by an m_0-CPDA.

The idea is that a configuration (M, ρ, k) will be represented by a state (M, k) of the machine and ρ will be encoded by the higher-order stack. Since M is a subterm of M_0 and k is the number of arguments M has, there are only finitely many states.

The alphabet Γ of the stack of the CPDA will contain elements of the form

$$(x, \gamma_i),\ (\gamma_i, N),\ (\gamma_i, \bot), \text{ for } i = 1, \ldots, max, \quad (x, N) \quad \text{and} \quad sp_l \text{ for } l = 1, \ldots, m_0.$$

Here x is a normal variable, γ_i is a stack variable, N is a subterm of M_0, and sp_l are special symbols whose only purpose is to make the stack pointers explicit in order to facilitate the presentation. The meaning of an element (x, γ_i) is that the value of the variable x is the same as the value of the stack variable γ_i, that in turn is determined by the rest of the stack. Symbols (x, N) and (γ_i, N) respectively say that the value of x and γ_i is N with the environment determined by the stack up to this symbol. Normally the values will be found on the topmost order 1 stack. But for stack variables we will sometimes need to follow a stack pointer sp_l. To define this precisely, we will need two auxiliary functions $val(z, S)$ and $find(\gamma_i, S)$ that, given a variable z, a stack variable γ_i and a stack S return a pair (N, S') of a term N and a stack S' that represent respectively the value of z and the

value of γ_i in S.

$$val(z,S) = \begin{cases} find(\gamma_i, pop_1(S)) & \text{if } top(S) = (z,\gamma_i) \text{ for some } i \\ (N, pop_1(S)) & \text{if } top(S) = (z,N) \\ val(z, pop_1(S)) & \text{otherwise.} \end{cases}$$

$$find(\gamma_i,S) = \begin{cases} (N, pop_1(S)) & \text{if } top(S) = (\gamma_i, N) \\ find(\gamma_i, collapse(S)) & \text{if } top(S) = sp_l \text{ for some } l \\ find(\gamma_i, pop_1(S)) & \text{otherwise.} \end{cases}$$

The first function traverses the top-most order 1 stack looking for a pair determining the value of the variable z. In case this value is a stack variable, the second function is called to get to the real value of the variable. The function *find* looks for a definition of γ_i. If it finds on the top of the stack a pair (γ_i, N), it returns N as a value of γ_i with the environment that is represented by the stack just below. If on the top of the stack it sees an sp_l pointer then it means that it should do *collapse* to search for the value. Observe that the index l is not used; it is there to simplify the proof of the correctness. If none of these cases holds then the *find* function continues the search on the top-most 1-stack.

With the help of these two functions it is straightforward to specify the environment $\rho[S]$ determined by S:

Definition 21. A stack S determines an environment $\rho[S]$ as follows:

$$\rho[S](x) = (N, \rho[S']) \quad \text{if } (N,S') = val(x,S) \text{ and } x \text{ is a normal variable.}$$
$$\rho[S](\gamma_i) = (N, \rho[S']) \quad \text{if } (N,S') = find(\gamma_i,S) \text{ and } \gamma_i \text{ is a stack variable.}$$

Observe that $\rho[S]$ is a partial function.

The following simple observation is the central place where the *collapse* operation is used. Since the *val* and *find* functions use only the pop_1 and *collapse* operations, the environment represented by a stack is not affected by the *copy* operations.

Lemma 22. For every $d = 2,\ldots,m$: if $S' = copy_d(S)$ then $\rho[S] = \rho[S']$.

Example 6. We start here a running example for this section. We consider the term

$$S = \big(Y f.\, \lambda g.\, g(f(\lambda z.gz))\big)\ (\lambda x.x)\ .$$

As in Example 1, we shorten the presentation by using the following abbreviations:

$$T = \lambda z.\, gz \qquad P = fT \qquad R = Y f.\, \lambda g.\, gP \qquad I = \lambda x.\, x \qquad S = RI\ .$$

The term S is producing a Böhm tree which is ω. Nevertheless, as we are interested in the relation between the computations of the Krivine machine and CPDA, it allows us to illustrate most of the interesting cases. We start here by showing in Figure 4 a third-order stack and the representation of the corresponding environment that is obtained according to Definition 21. The environment of the Krivine machine follows the graphical conventions we used in Example 1. The convention for high-order stacks is that the stacks of order 1, 3, . . . , 2n+1,. . . , grow upward while the stacks of order 2, 4, . . . , 2n, . . . grow

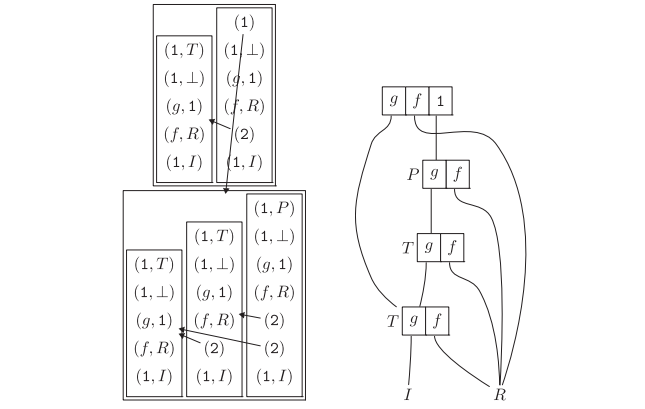


Fig. 4. Higher-order stack and the corresponding environment.

rightward. Moreover, for the sake of space and readability, we write (1), (2), (3), …, for sp_1, sp_2, sp_3, …, $(x,1)$, $(x,2)$, $(x,3)$,… for (x,γ_1), (x,γ_2), (x,γ_3), … and $(1,N)$, $(2,N)$, $(3,N)$,… for (γ_1,N), (γ_2,N), (γ_3,N),… Since only stack variables in the root environment can ever be accessed by an execution, we will never display stack variables in other nodes.

Now we describe the behaviour of the CPDA simulating the stackless Krivine machine.

Definition 23. Let M_0 be a closed term M_0 and $m_0 = comp(M_0) + 1$. For a variable z of M_0, we define its *link order* by $ll(z) = m_0 - order(z) + 1$.

We define the m_0-CPDA $\mathcal{A}(M_0)$ whose states are pairs (M,k), where M is a subterm of M_0 and k the number of its arguments (or equivalently, the arity of its type). The stack S of the CPDA will represent the environment $\rho[S]$ as described above. We define the behaviour of the CPDA by cases depending on the form of its state. In most cases, they simulate directly the computation rules of the Krivine machine from page 22.

— In a state (z,k):
If $ll(z) < m_0 + 1$,

$$\text{let} \quad (N,S') = val(z, copy_{ll(z)}(S)), \quad \text{and} \quad S'' = push^{sp_{order(z)},ll(z)}(S'),$$

the automaton changes its state to (N,k) and its stack to S''.
If $ll(z) = m_0 + 1$ then,

$$\text{let} \quad (N,S') = val(z,S) \quad \text{and} \quad S'' = S',$$

the automaton changes its state to (N,k) and its stack to S''. These operations implement the search for a value of the variable inside the higher-order stack. The copy operation is necessary to preserve the arguments of z. In the special case when $ll(z) = m_0 + 1$, the variable z has type 0 so it has no arguments and we do not need to do a *copy* operation.

— In a state $(\lambda x.M,k)$

$$\text{let} \quad S' = push^{(x,\gamma_k),1}(S) \quad \text{and} \quad S'' = push^{(\gamma_k,\perp),1}(S'),$$

the new state is $(M, k-1)$ and the new stack is S''. These two operations implement the assignment to x of the value at the top of the stack: this value is pointed by γ_k. Then the value of γ_k is set to undefined.

— In a state (MN, k)

$$\text{let} \quad S' = push^{(\gamma_{k+1},N),1}(S),$$

the state becomes $(M, k+1)$ and the stack S'. So M becomes the head term and the variable γ_{k+1} gets assigned $(N, \rho[S])$.

— In a state $(Yx.M, k)$

$$\text{let} \quad S' = push^{(x,Yx.M),1}(S),$$

the state becomes (M, k) and the stack S'. So M becomes the head term, and in the environment x is bound to the fixpoint term.

— In a state (b, k) with b a constant from Σ of arity k the automaton executes the transition $(b, qf_k, \ldots, qf_1)$. From a state qf_i and stack S it goes to $((N_i, 0), S_i)$ where $(N_i, S_i) = find(\gamma_i, S)$. This move implements outputting the constant and going to its arguments (cf. definition of $CTree(\mathcal{A})$ on page 14).

Let us comment on this definition. The case of (z, k) is the most complicated. Observe that if $order(z) = m_0 - 1$ then $ll(z) = 2$, and if $order(z) = 0$ then $ll(z) = m_0 + 1$. The goal of the copy operation is to preserve the meaning of stack variables. The later *push* makes a link to the initial higher-order stack where the values of stack variables can be found. More precisely we have that if we do $S_1 = copy_{ll(z)}(S)$ followed by $S_2 = push^{a,ll(z)}(S_1)$ and $S_3 = collapse(S_2)$ then $S_3 = S$; in other words we recover the original stack. We will prove later that the operation $val(z, \ldots)$ destroys only the part of the stack of order $< ll(z)$.

Observe that apart from the variable case, the automaton uses just the *push* operation.

For the proof of correctness we will need one more definition. We define the *argument order* of a higher-order stack S to be the maximal order of types of elements assigned to stack variables (recall that *max* is the maximal index of a stack variable).

$$ao(S) = \max\{order(N_i) : \rho[S](\gamma_i) = (N_i, \rho_i), \text{ and } i = 1, \ldots, max\}.$$

We are going to show that the CPDA defined above simulates the computation of the Krivine machine step by step. In the proof, we will show that the stack S of the CPDA satisfies the following invariant:

For every element sp_l in S: (*Inv*)

(i) the collapse pointer at sp_l is of order $d = m_0 - l + 1$, and

(ii) $l > ao(S')$ where S' is the part of S up to this element sp_l.

This property says that the subscript l of the sp_l symbol determines the order of the collapse pointer, and that, moreover, l is strictly greater than the orders of the stack variables stored on the stack obtained by following this pointer. This means that the orders of these stack variables give an upper bound on the collapse order d since d depends inversely on l. In other words, for every variable z: the meaning of z is on the topmost $ll(z)$ stack. This is a very important property as it will ensure that we will not

destroy a stack too much when we look for the value of a variable. The following lemma states this property formally. It is the central point in the correctness proof.

Lemma 24. Let S be a stack satisfying the condition (*Inv*). For every variable z, such that $\rho[S](z)$ is defined, every collapse operation that is used during the computation of $val(z,S)$ has order strictly smaller than $ll(z)$.

Proof. Let us examine the behaviour of the *val* function. First, the *val* function does a sequence of pop_1 operations until it gets to a pair (z,N) for some N or a pair (z,γ_i) for some γ_i. In the first case, since no collapse operation is used to compute $val(z,S)$, the lemma trivially holds. In the second case, we have that $order(\gamma_i) = order(z)$. The operation *find* is then started. This operation does pop_1 operations until it sees either a pair (γ_i,N), in which case the lemma holds trivially, or until it sees sp_l on the top of the stack; at that moment it does *collapse*. Suppose that it does *collapse* on a stack S_1. We know that the value of γ_i is defined for S_1 since it is defined for S and S_1 is an intermediate stack obtained when looking for the value of γ_i. Hence, by (*Inv*)(ii), $l > ao(S_1) \geqslant order(z)$. By the invariant (*Inv*)(i) the collapse done on S_1 is of order $m_0 - l + 1 < m_0 - order(z) + 1 = ll(z)$. Repeating this argument, we see that during the *val* operation the only stack operations are pop_1 and *collapse* of order smaller than $ll(z)$. □

We are ready to prove that the CPDA simulates the stackless Krivine machine.

Lemma 25. Let $(M,\rho,k) \to (M',\rho',k')$ be two successive configurations of the stackless Krivine machine. Let S be a higher order stack satisfying the condition (*Inv*) and $\rho[S] = \rho$. From the state (M,k) and the stack S the CPDA $\mathcal{A}(M_0)$ reaches the state (M',k') with a stack S' satisfying (*Inv*) condition and $\rho' = \rho[S']$.

Proof. The only case that is not straightforward is that of variable access. We have:

$$(z,\rho,k) \;\to\; (N,\rho'',k) \quad \text{where } (N,\rho') = \rho(z) \quad \text{and } \rho'' = \rho'[\gamma_1/\rho(\gamma_1),\ldots,\gamma_{max}/\rho(\gamma_{max})].$$

There are two cases to examine. The simpler one is when $order(z) = 0$. In this case $k = 0$, and in consequence $\rho'' = \rho'$. So we do not need to update stack variables. By hypothesis the CPDA is in the state $(z,0)$ and $\rho = \rho[S]$. By definition, the CPDA goes to state $(N,0)$ with the stack S' with $(N,S') = val(z,S)$. Since $\rho(z) = \rho[S](z)$, by the definition of $\rho[S](z)$ we have $\rho[S'] = \rho'$, and we are done.

Now consider the case when $order(z) > 0$. By hypothesis the CPDA is in the state (z,k) and $\rho = \rho[S]$. We recall the operations of the stack machine that are performed in this case, the automaton goes into the configuration whose state is (N,k) and whose stack is S'' where:

$$(N,S') = val(z,copy_{ll(z)}(S)), \quad \text{and} \quad S'' = push^{sp_{order(z)},ll(z)}(S').$$

By Lemma 22 and the assumption of the present lemma we have that $\rho[copy_{ll(z)}(S)] = \rho[S] = \rho$. In particular, $\rho(z) = \rho[S](z)$. By definition of $\rho[S]$ we have $(N,\rho') = (N,\rho[S'])$. Once again by definition of $\rho[S']$, the meaning of every normal variable in $\rho[S']$ is the same as in $\rho[S'']$.

We need to verify that the meaning of every stack variable is the same in $\rho[S'']$ and in $\rho[S]$. By definition $\rho[S''](\gamma_i) = (N_i, S_i)$ where $(N_i, S_i) = \mathit{find}(\gamma_i, S'')$. Then looking at the meaning of *find* we have $\mathit{find}(\gamma_i, S'') = \mathit{find}(\gamma_i, \mathit{collapse}(S''))$. But then we have $\mathit{collapse}(S'') = S$ since by Lemma 24 the value operation can destroy only the part of the topmost $ll(z)$-stack.

It remains to show that condition (*Inv*) holds. The first part of the condition follows from the definition as $ll(z) = m_0 - \mathit{order}(z) + 1$. The second part of the condition is clearly satisfied by S' since it is preserved by the *copy* operation. Next we do $S'' = \mathit{push}^{sp_{\mathit{order}}(z), ll(z)}(S')$. By the above, we have that $\mathit{pop}_{ll(z)}(S') = S$, so $ao(S'') = ao(S)$. Now since the stack variables on the stack are the arguments of z we have that for all i it holds that $\mathit{order}(z) > \mathit{order}(N_i)$ where $(N_i, \rho_i) = \rho(\gamma_i)$. Since $\rho = \rho[S]$ we have $\mathit{order}(z) > ao(S)$. This shows the second condition. □

Theorem 26. Consider terms and automata over a tree signature Σ. For every term M_0 of type 0 there is a CPDA $\mathcal{A}$ such that $BT(M_0) = CTree(\mathcal{A})$. The order of $\mathcal{A}$ is $\mathit{comp}(M_0)+1$.

Proof. We let $m_0 = \mathit{comp}(M_0) + 1$. Using Lemma 5, we consider $KTree(M_0)$ instead of $BT(M_0)$. The tree $KTree(M_0)$ starts with the execution of a Krivine machine from $(M_0, \varnothing, \varepsilon)$. By Lemma 20, we can as well look at the execution of the stackless Krivine machine from $(M_0, \varnothing, 0)$. We take the automaton $\mathcal{A}(M_0)$ as defined above: by construction it is an m_0-CPDA.

The $CTree(\mathcal{A}(M_0))$ starts from the configuration consisting of the state $(M_0, 0)$ and stack $\perp$. It is clear that $\rho[\perp] = \varnothing$ and $\perp$ satisfies the condition (*Inv*). Repeated applications of Lemma 25 give us that either both trees consist of the root labelled with ω, or on the *KTree* side we reach a configuration (b, ρ, k) and on the *CTree* side a configuration $((b, k), S)$ with $\rho = \rho[S]$ and S satisfying *Inv*. By definitions of both trees, they will have b in the root and this root will have k subtrees. The ith subtree on the one side will be $KTree(N_i, \rho_i, 0)$ where $(N_i, \rho_i) = \rho(\gamma_{k+1-i})$. On the other side, it will be $CTree((N_i, 0), S_i)$ where $(N_i, S_i) = \mathit{find}(\gamma_{k+1-i}, S)$. We have by definition of $\rho[S]$ that $\rho[S_i] = \rho_i$. Since S_i is an initial part of the stack S it satisfies the condition (*Inv*) too. Repeating this argument *ad infinitum*, we obtain that the trees $KTree(M_0)$ and $CTree(\mathcal{A}(M_0))$ are identical. □

Example 7. Consider the term S as defined in Example 6. Figure 5 illustrates the computation of $\mathcal{A}(S)$, in parallel with that of the stackless Krivine machine. This allows us to observe how the CPDA simulates the stackless Krivine machine. The current state of the CPDA is written just above its stack. The first five configurations are consecutive steps of CPDA and illustrate in order: the rule for Y-abstraction, the rule for λ-abstraction, the rule for application and the rule for searching the value of a variable g with $ll(g) \leqslant m_0$. Finally, the two last configurations (below the double line) illustrate the rule for searching the value of a variable z when $ll(z) = m_0 + 1$.

4.3. *Lowering the order*

If we start from a recursive scheme of order m, the translation from Lemma 12 will give us a term with complexity m (cf. Definition 2). So the construction from Theorem 26

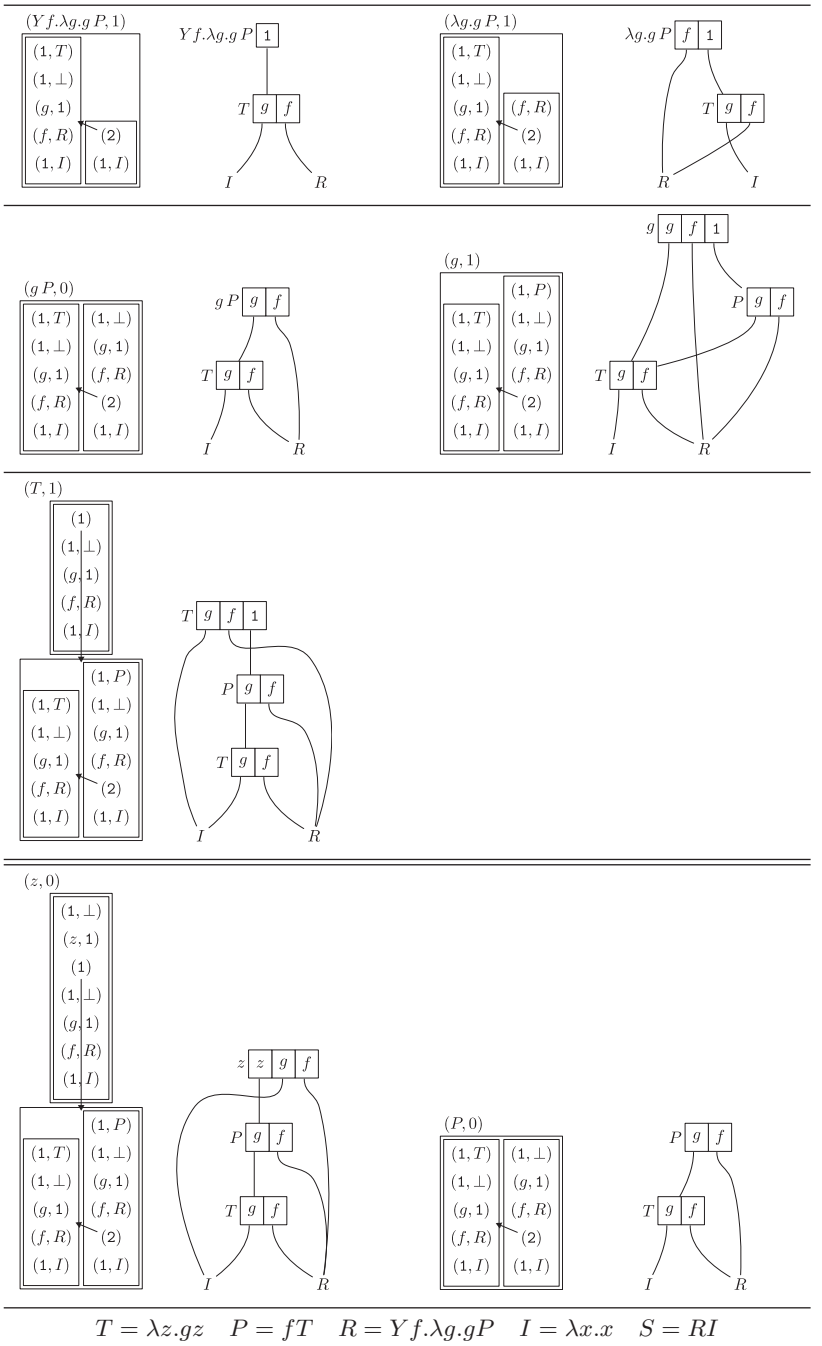


Fig. 5. Simulation of the stackless Krivine machine by $\mathcal{A}(S)$.

will produce a CPDA working with a stack of order $m+1$. Nevertheless, it is possible to produce an m-CPDA. For this, we adapt the previous method to terms that are in canonical form (cf Definition 15). We show how to obtain m-CPDA from a term M in a canonical form with $vcomp(M) = m$. Recall that every term of complexity m can be converted to a term in canonical form and of variable complexity at most m (Lemmas 3 and 17). By Lemma 3 we know that *vcomp* is a more precise measure than *comp* for terms in canonical form. Recall also that a term obtained from a translation of a recursive scheme of order m is a term in canonical form with variable complexity m (Definition 15). Thus this new method that works on terms in canonical form allows to show that m-CPDA can compute the trees of λY-terms of variable complexity m and of schemes of order m.

As we did when we introduced terms in canonical form, we assume that variables are partitioned between λ-variables and Y-variables and we write the latter in a boldface font. We show below that this distinction allows one to store only λ-variables in the environment. Since by Lemma 16 they have strictly smaller order than the Y-variables, we get the desired optimization.

We assume that the machine starts in a configuration $(M, \varnothing, \varepsilon)$ where M is a term in canonical form with pairwise distinct Y-variables. Recall also that we use $term(\mathbf{x})$ to denote the subterm starting with the binder of $\mathbf{x}$. We replace the rule for the Y-binder of the stackless Krivine machine (cf. page 22) with two new rules to cater for recursive variables:

$$\begin{aligned}(Y\,\mathbf{x}.P, \rho, k) &\to (P, \rho, k)\\ (\mathbf{x}, \rho, k) &\to (term(\mathbf{x}), \rho, k) \qquad \mathbf{x} \text{ is a } Y\text{-variable.}\end{aligned}$$

The first rule replaces the original fixpoint rule: the difference is that the value of the Y-variable $\mathbf{x}$ is no longer stored in the environment. The second rule tells us what to do when we encounter a Y-variable; this is needed since $\mathbf{x}$ will not appear in the environment. It is not difficult to see that for terms in canonical form the two rules faithfully simulate the original Krivine machine: we simply store only λ-variables in the environment. The environment is left unchanged in the first rule. This is harmless as all variables that are free in P are Y-variables. This allows us to implement this rule in the CPDA as a simple state change with no modification of the stack.

There is one more small special case that requires particular handling. For a term of complexity 0 we would like to obtain CPDA of order 0, that is a finite automaton. For this we need to forbid the use of the rule for application for configurations of the form $(aN_1 \dots N_p, \rho, S)$.

It is easy to adapt the construction of a CPDA from Definition 23 to the new rules. The first rule is simulated by the change of state from $(Y\,\mathbf{x}.P, k)$ to (P, k) without any change to the stack. Similarly, the second new rule is simulated by the change of state from $(\mathbf{x}, k)$ to $(term(\mathbf{x}), k)$. Finally, when the state is of the form $(bN_1 \dots N_p, k)$, the CPDA performs the transition $(b, N_1, \dots, N_p)$. It is straightforward to check that Lemma 25 still holds for the modified Krivine machine and for the modified CPDA.

To sum up, starting with a term M of complexity m, Lemma 17 tells us how to obtain a term M' in the canonical form so that $BT(M') = BT(M)$ and $vcomp(M') \leqslant comp(M') \leqslant m$. As M' is in the canonical form and has variable complexity m, by Lemma 16 the only variables that are of order m are Y-variables. In our modified translation, the CPDA stores in its stack only λ-variables, and these have order at most $m-1$. Hence, in Definition 23 we can take $m_0 = m$. We obtain m-CPDA equivalent to M.

In conclusion, if one is only interested in translating recursive schemes to CPDA then the translation from the previous section can be taken as it is with the exception that nonterminals are never stored in the environment, as their value is just given by the scheme. This is summarized in the following theorem.

Theorem 27. Consider terms and automata over a tree signature Σ. For every term M of type 0 there is a CPDA $\mathcal{A}$ such that $BT(M) = CTree(\mathcal{A})$. Moreover the order of $\mathcal{A}$ is $vcomp(M)$. In particular, if M is obtained by the translation (cf. Theorem 19) of a recursive scheme of order m then $\mathcal{A}$ is also of order m.

Example 8. So as to illustrate this new way of evaluating λY-terms we show in Figure 6 the configurations of the CPDA computing the Böhm tree of the term used in Examples 6 and 7. The transitions in Figure 6 are the same as the ones in Figure 5. One can remark that, when compared to the CPDA presented in Figure 5, the order of the stack is indeed lowered by 1.

4.4. *Relation to safety*

As we mentioned in the introduction, the safety restriction has played an important role in the study of higher-order recursive schemes. At first, Damm (1982) studied higher-order recursive schemes using implicitly the restriction of safety. Indeed, Damm worked with the notion of *derived types*, which are types of the form $\alpha_n \to \cdots \to \alpha_1 \to o$ where the types α_i are products (possibly empty) of derived types and $order(\alpha_{i+1}) = order(\alpha_i) + 1$. Actually, it is rather easy to show that every type is isomorphic to a derived type so this form of types is not a restriction in itself. When one considers only schemes defined with applicative terms though, the requirement that all the types need to be derived types makes the scheme safe (Miranda 2006). In his original definition, Damm permits the use of λ-abstraction in schemes but then wrongly claims in Corollary 4.12 of his paper that every scheme can be converted to an applicative scheme using only derived types. The flaw in Damm's proof of Corollary 4.12 stems from the unsoundness of the procedure he proposes to eliminate λ-abstractions; in particular the problem is situated at the transformation of schemes (with λ-abstraction) into what he calls *locally closed* schemes. In consequence, Damm's work is about safe schemes even though restriction to safety in his work is implicit due the restrictions to applicative schemes. A recent result of Parys (2012) shows that safety is indeed a restriction in a sense that there exist schemes that are not equivalent to a safe scheme. The notion of safe schemes has been made explicit by Knapik *et al.* (2002) who identified the class of *safe recursive schemes*. They have shown that this class defines the same set of infinite trees as higher-order pushdown automata. The latter are as CPDA but without the *collapse* operation.

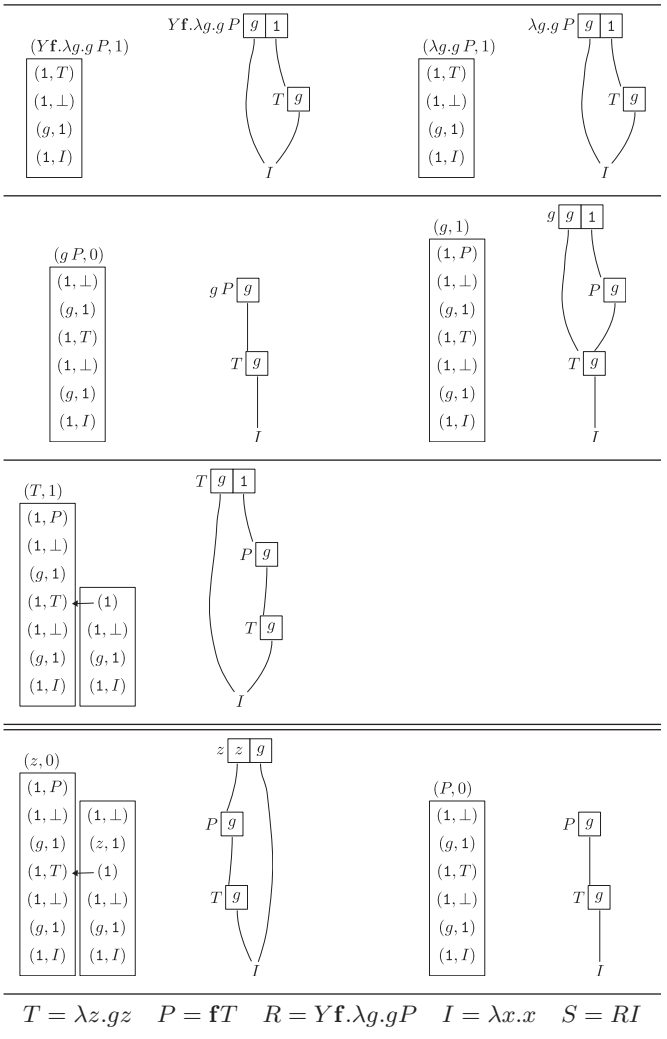


Fig. 6. Simulation of the stackless Krivine machine that reduces the order.

In this section, we will examine how our translation behaves with respect to safe schemes. In particular, we are going to see that the *collapse* operations that are performed during the execution of the CPDA can be eliminated: indeed, the depth of the collapse pointer will always be 1.

It is worth noticing that Blum and Ong (2009) have extended the notion of safety from Knapik *et al.* (2002) to the simply-typed λ-calculus in a clear way. Nevertheless, the straightforward adaptation of their notion of safety to PCF does not yield a satisfactory notion of *safe* λY-calculus. In particular, there is no simple translation of safe schemes into this safe λY-calculus. In order to remedy this the first step is to once again make the distinction between λ-variables and Y-variables; hence we assume that we are given

two disjoint sets of variables λ-variables and Y-variables. We write $nFV(M)$ for the set of free λ-variables of M.

Definition 28. A term M is *superficially safe* when for every x in $nFV(M)$, $order(M) \leqslant order(x)$.

A term M is *almost safe* when:

1. if $Y\mathbf{x}.N$ is a subterm of M then all the free variables in N are Y-variables.
2. if PQ is a subterm of M, then Q is superficially safe and if P is not an application (i.e. P is not of the form P_1P_2) then P is superficially safe.

A term M is *safe* when it is both almost safe and superficially safe.

A scheme is *safe* when it associates to each non-terminal F a safe term $\lambda\overrightarrow{x}.M$ (where M is an applicative term).

Observe that when M is almost safe and contains a subterm PQ, the definition implies that Q is at the same time superficially safe and almost safe, so that Q needs to be safe. Moreover, when P is not an application, P is also safe for the same reasons.

The reason why we need to make this distinction between λ-variables and Y-variables comes from the fact that, without this distinction, it would be unclear how to translate safe schemes into safe λY-terms. This distinction is actually present implicitly in the definition of safety for schemes we have given and which is equivalent to the one given by Knapik *et al.* (2002), where, contrary to parameters (λ-variables in our setting), the nonterminals of schemes (Y-variables in our setting) are not taken into account. Indeed let us consider a safe scheme where the non-terminal S is of type 0 and the non-terminal F is of type $0 \to 0$:

$$\begin{aligned} S &\equiv F\,a \\ F &\equiv \lambda x.F(b\,S\,x). \end{aligned}$$

This scheme would be translated into the term $Y\,S.\,Ma$ where $M = Y\,F.\,\lambda x.\,F(b\,S\,x)$. But $Y\,S.\,Ma$ is safe only because the unique free variable in M, namely S, is a Y-variable. If S were a λ-variable, then, as S is of order 1 and M of order 2, M would not be superficially safe and thus $Y\,S.\,Ma$ would not be safe.

Now, it is easy to see that the translation of a safe scheme into a λY-term results in a safe term and that conversely the translation of a safe λY-term (not necessarily in the canonical form) results in a safe scheme (cf. Section 3.1).

As in the previous section, we are going to work with terms in canonical form. This will allow us not to store recursive variables when we will evaluate safe terms. We now extend the notion of safety to configurations of stackless Krivine machines.

Definition 29. An *environment ρ is safe* when for every x in its domain $\rho(x) = (N, \rho')$ is a *safe closure*, that is N is a safe term and ρ' is a safe environment.

A *configuration of a stackless Krivine machine* (M, ρ, k) *is safe* when M is almost safe, and ρ is a safe environment, and for every i in $[1, k]$, $\rho(\gamma_i)$ is a safe closure.

As expected, safe configurations are closed under reduction (when we take the rule of Section 4.3 for the execution of the recursion on Y-binders). The proof of the lemma follows by a direct case inspection.

Lemma 30. If (M,ρ,k) is safe and $(M,\rho,k) \to (M',\rho',k')$, then (M',ρ',k') is also safe.

Let us examine the behaviour of a CPDA obtained from the translation of a safe term (Definition 23). Recall that every letter on a higher-order stack has a superscript of the form (i,k) determining the collapse link. The component i designates the type of the link, that is the level of the *pop* operation to be performed. The component k is the depth of the link, that is how many pop operations should be performed. Clearly, a *collapse* operation on a link of depth 1 can be just replaced by a single *pop* operation. We will show that the CPDA obtained from the translation of a safe scheme performs the *collapse* operation only on links of depth 1 (this does not mean that the stack may not contain links of greater depth; this means that when the *collapse* operation is applied, the link has depth 1).

We will say that a *link in a stack S is active from a term N* if either: (i) it is collapsed by the $val(x,S)$ operation for some $x \in nFV(N)$, or (ii) it is active in S' from N', where $(N',S') = val(x,S)$ for some $x \in nFV(N)$. Similarly a *link in a stack S is active from a stack variable γ_i* if it is traversed by $find(\gamma_i,S)$, or if it is active in S' from N', where $(N',S') = find(\gamma_i,S)$. Finally, we say that a *link is active in S from a state* (N,k) if it is active from N or from γ_i for some $i = 1,\ldots,k$.

It should be clear that a CPDA obtained from our translation traverses only active links. We show that if we start from a safe λY-term in the canonical form then all the active links have depth 1. This is a consequence of the following specialization of Lemma 25 to the safe case.

Lemma 31. Let $(M,\rho,k) \to (M',\rho',k')$ be two successive safe configurations of the stackless Krivine machine. Let S be a higher order stack such that $\rho[S] = \rho$, all active links from (M,k) are of depth 1, and the condition (*Inv*) is satisfied. From the state (M,k) and the stack S the CPDA $\mathcal{A}(M_0)$ reaches the state (M',k') and a stack S' with all links in S' active from (M',k') having depth 1.

Proof. As for Lemma 25, the only non-trivial case is that of variable access. If $order(z) = 0$ then S' is obtained by the *val* operation so the lemma is immediate. Suppose that $order(z) > 0$. Let us recall the operations of the CPDA that are performed in this case. It goes into the configuration $((N,k),S'')$ where:

$$(N,S') = val(z,copy_{ll(z)}(S)), \quad \text{and} \quad S'' = push^{sp_{order(z)},ll(z)}(S').$$

By Lemma 24 the links traversed by the operation $val(z,copy_{ll(z)}(S))$ are all of order strictly smaller than $ll(z)$. Therefore, all the operations performed during the operation $val(z,copy_{ll(z)}(S))$ are local to the topmost stack of order $ll(z)$. We thus have that

$$pop_{ll(z)}(val(z,copy_{ll(z)}(S))) = S \text{ and therefore, } collapse(S'') = S.$$

We are ready to show that all active links are of depth 1. We first consider links from γ_i for i in $[1,k]$. The computation of $find(\gamma_i,S'')$ starts with the *collapse* operation, because

the topmost symbol of S'' is $sp_{order(z)}$. As a consequence, a link that is active in S'' from γ_i is either the link of the topmost symbol $sp_{order(z)}$ which has depth one, or a link that is active from γ_i in S which, by hypothesis, has depth one.

Consider now links active from N. Take a variable $y \in nFV(N)$. Since N is a safe term $order(y) \geqslant order(N) = order(z)$. By a repeated use of Lemma 24, we show that all active links in S'' from y are of order strictly smaller than $ll(z)$. But then, all those links are in the topmost stack of order $ll(z)$ of S'' that is created, before the *push* operation, by the operation $copy_{ll(z)}$ applied to S. So, since they were already active from z in S, the depth of those links is one in the topmost stack of order $ll(z)$ of S. They also have depth one in S'' since their order is strictly smaller than $ll(z)$ and since the operation $copy_{ll(z)}$ only increases the depth of links of order $ll(z)$. □

By combining this lemma with Theorem 27 we obtain the following:

Theorem 32. Consider terms and automata over a tree signature Σ. For every safe term M of type 0 there is a higher-order pushdown automaton without collapse $\mathcal{A}$ such that $BT(M) = CTree(\mathcal{A})$. The order of $\mathcal{A}$ is $vcomp(M)$. In particular, if M is obtained by the translation (cf. Theorem 19) of a safe recursive scheme of order m then $\mathcal{A}$ is also of order m.

5. Another translation

We now present another translation from λY-calculus to CPDAs. This translation is different in the way it handles the stack of the Krivine machine: by using tuples of terms, the stack will contain at most one element. The translation also differs in the way closures are represented on the higher-order stack, and the way the values of the variables are retrieved by the CPDA. This translation is a bit more technical to define but then it is slightly easier to prove its correctness.

In the first subsection, we will use some standard transformations of terms to put them into a convenient form. In particular, we will introduce product types and use them to uncurry terms. This will permit to have stacks of size at most 1 in the configurations of Krivine machine. The second subsection will give a translation working on so prepared terms, and using the specialized version of the Krivine machine.

5.1. *Uncurrying of λ-terms and Krivine machines with product*

For this translation, we need to start with a term in η-long form and in canonical form (see Section 2.1 and Definition 15). We assume that variables are partitioned between λ-variables and Y-variables and, as we did before, we shall write the latter in a boldface font. We then apply a syntactic transformation to terms called *uncurrying*. With this transformation, we group every sequence of arguments of a term into a tuple. A consequence is that every term has at most one argument. To do this, we enrich simple types with a *finitary product*: given types $\alpha_1, \ldots, \alpha_n$, we write $\alpha_1 \times \cdots \times \alpha_n$ for their product. The counter-part of product types in the syntax of the λ-calculus is given by the possibility of constructing tuples of terms and to apply projections to terms. Formally,

given terms $M_1, \ldots, M_n$ respectively of type $\alpha_1, \ldots, \alpha_n$, the term $\langle M_1, \ldots, M_n \rangle$ is of type $\alpha_1 \times \cdots \times \alpha_n$. Moreover, given a term M of type $\alpha_1 \times \cdots \times \alpha_n$ and given i in $[1, n]$, the term $\pi_i(M)$ is of type α_i. Finally, we extend β-reduction with the rule: $\pi_i(\langle M_1, \ldots, M_n \rangle) \to_\beta M_i$. As a shorthand we may write $\overrightarrow{N_p}$ instead of $\langle N_1, \ldots, N_p \rangle$. To reduce the number of cases in definitions and proofs, we also consider terms of type α as one-dimensional tuples when α is not a type of the form $\alpha_1 \times \cdots \times \alpha_n$. Thus for a term N that has such a type α, the notations $\pi_1(N)$, $\langle N \rangle$ and $\pi_1(\langle N \rangle)$ simply denote N.

The notion of order of a type is adapted to types with products as follows: $order(0) = 0$, $order(\gamma \to \alpha) = \max(order(\gamma) + 1, \alpha)$ and $order(\alpha_1 \times \cdots \times \alpha_n) = \max_{i \in [1;n]}(order(\alpha_i))$. The order of a term is the order of its type.

We now define an operation *unc* that transforms types into uncurried types and η-long terms in canonical form into uncurried terms:

1. $unc(0) = 0$,
2. $unc(\alpha_1 \to \cdots \to \alpha_n \to 0) = (unc(\alpha_1) \times \cdots \times unc(\alpha_n)) \to 0$,
3. $unc((Y\,\mathbf{x}^\alpha.P)N_1 \ldots N_m) = (Y\,\mathbf{x}^{unc(\alpha)}.unc(P[\mathbf{x}^{unc(\alpha)}/\mathbf{x}^\alpha]))\langle unc(N_1), \ldots, unc(N_m)\rangle$,
4. $unc(\lambda x_1^{\alpha_1} \ldots x_n^{\alpha_n}.P) = \lambda z^\gamma.unc(P[\pi_1(z^\gamma)/x_1^{\alpha_1}, \ldots \pi_n(z^\gamma)/x_n^{\alpha_n}])$ where P has type 0 and $\gamma = unc(\alpha_1) \times \cdots \times unc(\alpha_n)$,
5. $unc(\pi_i(z^\gamma)N_1 \ldots N_m) = \pi_i(z^\gamma)\langle unc(N_1), \ldots, unc(N_m)\rangle$,
6. $unc(\mathbf{x}^\alpha N_1 \ldots N_m) = \mathbf{x}^\alpha\langle unc(N_1), \ldots, unc(N_m)\rangle$,
7. $unc(bN_1 \ldots N_m) = b\,unc(N_1) \ldots unc(N_m)$.

As the transformation is applied to terms in η-long form, rule 4 is sufficient to deal with all λ-abstractions. Let us also note that the operation *unc* produces intermediate results that are not well typed; for example, in rules 3 and 4 it substitutes variables with uncurried types for original variables, leading to a term that is not well typed. Of course, in the end, the transformation *unc* produces a well-typed term; so as to avoid this drawback, we could have used explicit substitutions in a manner akin to the Krivine machine, but we have preferred a more direct presentation of the transformation.

Notice that for an uncurried type, $\gamma \to 0$, we have $order(\gamma \to 0) = order(\gamma) + 1$. As we have already mentioned, the role of uncurrying is to make the stack of the Krivine machine simple. Indeed with an uncurried term, the stack contains at most one element. In the translation into CPDA, this makes it easier for the CPDA to retrieve the closure bound to a variable. In a certain sense the role played by the stack variables γ_i in the previous translation is now fulfilled by the projections π_i. Observe that we have not uncurried the type of constants, so that, for a closed term M of type 0, the Böhm tree of $unc(M)$ is the same as the Böhm tree of M.

Observe that as we have started with an η-long term in canonical form, the term M we have obtained after uncurrying has the property that every subterm of the form $Y\,\mathbf{x}.P$ has as free variables only Y-variables. Recall that we use $term(\mathbf{x})$ to denote the subterm starting with the binder of $\mathbf{x}$.

We now adapt the Krivine machine to computing the Böhm tree of terms of the form $unc(M)$. For this, we slightly change the notion of environments. In the definition of the Krivine machine in Section 2.2, we considered that environments were functions from variables to closures. Here, so as to emphasize the way we make CPDA retrieve the

closures associated with variables, we structure the environment of the Krivine machine as an association list realizing the mapping of a variable to its closure. The Krivine machine finds the closure associated to a variable by scanning the list representing the environment. The mutually recursive definition of closures and environments is thus now:

$$C ::= (M, \rho) \qquad \rho ::= \varnothing \mid [x \mapsto C] :: \rho \ ,$$

where :: denotes the operation of appending an element as the head of a list. As a shorthand, we may write $\rho(x)$ to denote the first closure bound by x in the environment ρ. The rules of the Krivine machine are now:

$$\begin{aligned}
(M\overrightarrow{N_p}, \rho, \varepsilon) &\rightarrow (M, \rho, (\overrightarrow{N_p}, \rho)) \text{ when } M \text{ is not a constant} \\
(\lambda x.M, \sigma, (\overrightarrow{N_p}, \rho')) &\rightarrow (M, [x \mapsto (\overrightarrow{N_p}, \rho')] :: \rho, \varepsilon) \\
(\pi_i(x), [x \mapsto (\overrightarrow{N_p}, \rho')] :: \rho, \sigma) &\rightarrow (N_i, \rho', \sigma) \\
(\pi_i(x), [y \mapsto (\overrightarrow{N_p}, \rho')] :: \rho, \sigma) &\rightarrow (\pi_i(x), \rho, \sigma) \text{ when } x \neq y \\
(\mathbf{x}, \rho, \sigma) &\rightarrow (term(\mathbf{x}), \rho, \sigma) \\
((Y\,\mathbf{x}.P), \rho, \sigma) &\rightarrow (P, \rho, \sigma).
\end{aligned}$$

So as to make precise the relationship between this version of the Krivine machine computing the Böhm tree of uncurried terms, and the Krivine machine we have defined in Section 2.2, we define the notion of $KPTree(M, \rho, \sigma)$ of the Böhm tree computed by the Krivine machine on uncurried terms from the configuration (M, ρ, σ). If from the configuration (M, ρ, σ) the machine reaches a configuration $(bN_1 \dots N_l, \rho, \varepsilon)$ then $KPTree(M, \rho, \sigma)$ is the tree whose root is labelled b, and whose subtrees are (in that order) $KPTree(N_1, \rho, \varepsilon), \dots, KPTree(N_l, \rho, \varepsilon)$. Otherwise $KPTree(M, \rho, \sigma)$ is ω. Given a closed η-long term M of type 0 in canonical form, we write $KPTree(M)$ for $KPTree(unc(M), \varnothing, \varepsilon)$. The following lemma gives the expected equivalence result. We omit the proof which relies on the usual isomorphism between curried and uncurried terms.

Lemma 33. Given a closed η-long term M of type 0 in canonical form, we have that $KTree(M) = KPTree(M)$.

5.2. *Simulation*

We fix for this section a term M_0 in η-long and canonical form and we let $m_0 = vcomp(M_0)$. We are going to construct an m_0-CPDA $\mathcal{B}(M_0) = (\Sigma, \Gamma, Q, F, \delta)$ generating $KPTree(M_0)$ (hence, $BT(M_0)$ thanks to Lemmas 10 and 33). The main part of the construction of $\mathcal{B}(M_0)$ consists in representing the environment of the Krivine machine directly in the higher-order stack. The topmost 1-stack represents the sequence of variables that the environment is binding and the closure associated to those variables can be accessed using the *collapse* operation. In turn, a closure is represented by a higher-order stack whose top-element is the term of the closure while the environment of the closure is represented by the stack simply obtained by poping this top element. The λ-variables with the highest type order (of order $m_0 - 1$ by Lemma 16) have a special treatment: the closure they are bound to is represented in the same 1-stack, and this closure can be

retrieved simply using the pop_1 operation (or equivalently using a collapse on a link of order 1).

The stack alphabet Γ of $\mathcal{B}(M_0)$ is the set of λ-variables of M_0 together with tuples of terms that are arguments of some subterm of M_0. The set Q of states of $\mathcal{B}(M_0)$ is the set of subterms of M_0. Before we give the transition rules of $\mathcal{B}(M_0)$, we explain the way the environment of the Krivine machine is represented as a higher-order stack. We are going to define the function $val^\flat$ to make a parallel with the function val of the first translation. Here, instead of being applied to a variable and a higher-order stack it is applied to a variable being projected and a higher-order stack:

$$val^\flat(\pi_i(z), S) = \begin{cases} (N_i, pop_1(S')) & \text{if } top(S) = z,\ S' = collapse(S) \\ & \text{and } top(S') = \overrightarrow{N_l} \\ val^\flat(\pi_i(z), pop_1(S)) & \text{if } top(S) = y \neq z \text{ and } order(y) < m_0 - 1 \\ val^\flat(\pi_i(z), pop_1(pop_1(S))) & \text{if } top(S) = y \neq z \text{ and } order(y) = m_0 - 1 \end{cases}$$

Therefore to find the value associated to $\pi_i(z)$, it suffices do the following steps:

1. search in the topmost first-order stack the first occurrence of z with the pop_1 operation,
2. use the *collapse* operation to access the closure assigned to z,
3. use the index of the projection to select the appropriate term in the tuple on the top of the stack obtained after collapse,
4. and finally erase that tuple from the stack using the pop_1 operation to get the environment from the closure.

Notice that when we scan the topmost first order stack and we encounter a variable of order $m_0 - 1$ that we need to skip, then, this skipping requires one to use the pop_1 operation twice instead of just once as in the other cases. This is due to the fact that the closures associated to variables of order $m_0 - 1$ are represented in the same stack as the environment.

We are representing the environment of the Krivine machine as a sequence of variables bound to closures. Let us see how to retrieve this sequence from a higher-order stack. So given a higher-order stack S the environment associated to S, $\rho[S]$ is:

1. if $top(S) = x^\alpha$ then $\rho[S] = [x^\alpha \mapsto (\overrightarrow{N_l}, \rho_2)] :: \rho_1$ where:
   a. $\rho_1 = \begin{cases} \rho[pop_1(pop_1(S))] \text{ when } order(\alpha) = m_0 - 1 \\ \rho[pop_1(S)] \text{ otherwise,} \end{cases}$
   b. $\overrightarrow{N_l} = top(collapse(S))$,
   c. $\rho_2 = \rho[pop_1(collapse(S))]$.
2. $\rho = \varnothing$ otherwise.

Notice that, unsurprisingly, the way closures are constructed in $\rho[S]$ is similar to the definition of $val^\flat(\pi_i(x), S)$. The only difference is that the function $val^\flat$ could be used to compute the environment as a mapping from variables to closures while $\rho[S]$ computes the environment as an association list of variables and closures. Notice also that, similarly to the definition of $val^\flat(\pi_i(x), S)$, λ-variables of maximal order, that is of order $m_0 - 1$, receive a particular treatment. We shall comment on this with more details later on.

As in the previous translation, it is worth noticing that the operation $\rho[S]$ is insensitive to the operation $copy_d$ when $d > 1$, so we have $\rho[S] = \rho[copy_d(S)]$.

Definition 34. Let M_0 be a closed term of type 0. We suppose that M_0 is in the canonical and uncurried forms, and we let $m_0 = vcomp(M_0)$. We define an m_0-CPDA $\mathcal{B}(M_0)$ whose states are subterms of M_0 which are not tuples and whose stack alphabet are the variables of M_0 and the tuples that are in an argument position in M_0. The initial state of $\mathcal{B}(M_0)$ is M_0. We now describe the transitions of $\mathcal{B}(M_0)$ from a configuration (q, S):

1. When $q = \pi_i(z)$, there are two possibilities:
   a. In the case $top(S) = z$, the automaton goes to the configuration (N_i, S'') where $S'' = pop_1(S')$, $S' = collapse(S)$, $\overrightarrow{N_l} = top(S')$ and $N_i = \pi_i(\overrightarrow{N_l})$.
   b. In the case $top(S) = y$ with $y \neq z$, the automaton goes to the configuration $(\pi_i(z), S')$, where $S' = pop_1(S)$ when $order(y) < m_0 - 1$ and $S' = pop_1(pop_1(S))$ when $order(y) = m_0 - 1$.
2. When $q = \lambda x.M$, the automaton goes to the configuration (M, S') where $S' = push^{x,p}(S)$ and $p = m_0 - order(x)$.
3. When $q = M\overrightarrow{N_l}$ (M not being a constant), the automaton goes to the configuration (M, S''') where $S''' = pop_1(S'')$, $S'' = copy_p(S')$ and $S' = push^{\overrightarrow{N_l},1}(S)$ (where $p = m_0 - order(\overrightarrow{N_l})$).
4. When $q = \mathbf{x}$, $\mathbf{x}$ being a Y-variable, the automaton goes to the configuration $(term(\mathbf{x}), S)$.
5. When $q = Y\,\mathbf{x}.P$, the automaton goes to the configuration (P, S).
6. When $q = bN_1 \ldots N_l$ where b is a constant of arity l from the tree signature, the automaton goes to $(b, N_1, \ldots, N_l)$. This move implements outputting the constant and going to its arguments.

Let us briefly explain these rules.

The first rule looks up the value of the variable: it implements the operation $val^{\flat}$. If the variable is on the top of the stack we just recover the value by performing a *collapse* operation, as we have explained when describing the encoding of the environment in the stack. If the variable is not on the top of S then we must go deeper into the stack. The difference in treatment depending on the order of y comes from the fact that when y has order $m_0 - 1$, then the term of the closure y is bound to is the next stack symbol on the stack; and so as to advance to the next variable the automaton needs to get rid of the variable y and of the term y is bound to by using two consecutive pop_1 operations.

In a state $\lambda x.M$ the automaton implements the binding of a closure to the variable x. The fact that we use the operation $push^{x,p}$ requires that the closure has been prepared beforehand and is represented in the stack $pop_p(S)$ with $p = m_0 - order(x)$.

In a state $M\overrightarrow{N_l}$ (M not being a constant), we need to prepare the closure containing $\overrightarrow{N_l}$ with the current environment that is going to be bound to the variable abstracted in the term to which M is going to be evaluated. For this, it suffices to push $\overrightarrow{N_l}$ on top of S, and, so as to be consistent with the rule of the automaton dealing with λ-abstraction, use the operation $copy_p$ where $p = m_0 - order(\overrightarrow{N_l})$ and remove with pop_1 the topmost occurrence

of $\overrightarrow{N_l}$ created by the copy. Notice that when $order(\overrightarrow{N_l}) = m_0 - 1$, this operation simply results in pushing $\overrightarrow{N_l}$ on top of the topmost order 1 stack.

We can now say what it means for a configuration of the CPDA to represent a configuration of a Krivine machine:

Definition 35. We say that a configuration (P, S) of the CPDA *represents* a configuration (P, ρ, σ) of the Krivine machine if the following three conditions hold:

1. $\rho = \begin{cases} \rho[S] \text{ when } order(P) < m_0 \\ \rho[pop_1(S)] \text{ when } order(P) = m_0 \end{cases}$
2. When σ is nonempty then $\sigma = (\overrightarrow{N_l}, \rho[S''])$ where $S'' = pop_1(S')$, $\overrightarrow{N_l} = top(S')$ and:

$$S' = \begin{cases} pop_{m_0 - order(P)+1}(S) \text{ when } order(P) < m_0 \\ S \text{ when } order(P) = m_0 \end{cases}$$

3. For every variable x in S, the collapse pointer at x is of order $m_0 - order(x)$.

Let us now comment a bit on this definition that relates configurations of CPDA to configurations of the Krivine machine. When P has order one, then, because of typing, the stack of the Krivine machine needs to be empty. In case P has higher order, because the machine is used only to evaluate terms of type 0, the stack of the Krivine machine has to contain a closure. We have seen in the rules of the CPDA that, when P is of higher-order type $\alpha \to 0$, we have prepared the closure so that we can bind some variable x^α by applying a $push^{x^\alpha, p}$ operation with $p = m_0 - order(\alpha)$. But $order(\alpha) = order(P) - 1$ so that $p = m_0 - order(P) + 1$. Thus we shall be able to retrieve the closure, simply by applying a $pop_{m_0 - order(P)+1}$ operation. Nevertheless, this only works when $order(P) < m_0$; indeed if $order(P) = m_0$ then $p = 1$ and the binding is made with a $push^{x^\alpha, 1}$ operation, meaning that the closure is on top of the topmost 1-stack of the automaton. This explains why the case where $order(P) = m_0$ is treated differently.

Along with this observation we can make a further remark on the case where the state is $\pi_i(z)$ and $top(S) = z$. In this case, the automaton uses a *collapse* operation. As the variable z must have been pushed on the stack with a $push^{z,p}$ operation where $p = m_0 - order(z)$, the *collapse* operation is a sequence of pop_p operations. We must ensure that these pop_p operations do not destroy the closure that is on the stack of the Krivine machine. The closure has an order d strictly smaller than $order(z)$ as the closure is supposed to be bound by a λ-abstraction in the term $\pi_i(z)$. So the closure is accessible with $pop_{m_0 - d}$ operation. Since $m_0 - d > p = m_0 - order(z)$, we are guaranteed that, after the pop_p operations used to search the value of z, all the order $m_0 - d$ stacks that are contained in the original stack are also in the new stack.

We can now prove that $\mathcal{B}(M_0)$ simulates the Krivine machine in a similar manner as it was done in the previous translation.

Lemma 36. Given a configuration (P, S) of $\mathcal{B}(M_0)$ representing (P, ρ, σ). If $(P, \rho, \sigma) \to (P_1, \rho_1, \sigma_1)$ then from (P, S), $\mathcal{A}$ reaches a configuration (P_1, S_1) representing (P_1, ρ_1, σ_1).

Proof. The lemma is proved by case analysis on the shape of P. The proof proceeds by examining the definition. The only interesting case is the one where $P = \pi_i(z)$ and $top(S) = z$. Since (P, S) represents (P, ρ, σ), we need to have $\rho = (z, \overrightarrow{N_l}, \rho_2) :: \rho_1$ so that:

1. $\rho_1 = \begin{cases} \rho[pop_1(pop_1(S))] \text{ when } order(z) = m_0 - 1 \\ \rho[pop_1(S)] \text{ otherwise,} \end{cases}$
2. $\overrightarrow{N_l} = top(collapse(S))$,
3. $\rho_2 = \rho[pop_1(collapse(S))]$.

We thus have $(P_1, S_1) = (N_i, pop_1(collapse(S)))$ and $(P, \rho, \sigma) \to (N_i, \rho_2, \sigma)$. In case $\sigma = \varepsilon$ (i.e. z has type of order 0), then (P_1, S_1) obviously represents (N_i, ρ_2, σ). In case $\sigma \neq \varepsilon$, as the order of z is at most $m_0 - 1$, $order(P)$ is also at most $m_0 - 1$ and we have that $\sigma = (\overrightarrow{Q_k}, \rho[S''])$ where $S'' = pop_1(S')$, $\overrightarrow{Q}_k = top(S')$ and $S' = pop_{m_0 - order(P)+1}(S)$. But since the *collapse* operation on S is a sequence of $pop_{m_0 - order(z)}$ operations, and, since $order(P) \leqslant order(z)$, we have that $S' = pop_{m - order(P)+1}(S) = pop_{m - order(P)+1}(S_1)$. Thus, because $order(N_i) = order(P)$, we get

$$pop_{m - order(N_i)+1}(S_1) = pop_{m - order(P)+1}(S_1) = S'.$$

So (P_1, S_1) represents (N_i, ρ_2, σ). □

Theorem 37. Let M be a term of type 0 in the canonical form, let $unc(M)$ be the uncurried form of M. If $m = vcomp(M)$ then $\mathcal{B}(unc(M))$ is an m-CPDA such that $BT(M) = CTree(\mathcal{B}(unc(M))$.

Proof. Using Lemmas 10 and 33 we consider $KPTree(M)$ instead of $BT(M)$. The tree $KPTree(M)$ starts with the execution of a Krivine machine from $(unc(M), \varnothing, \varepsilon)$. On the other side $CTree(\mathcal{B}(unc(M)))$ starts from the configuration consisting of the state $unc(M)$ and the stack $\perp$. It is clear that $(unc(M), \perp)$ represents $(unc(M), \varnothing, \varepsilon)$. Repeated applications of Lemma 36 give us that either both trees consist of the root labelled with ω, or that the Krivine machine reaches a configuration $(bN_1 \dots N_l, \rho, \varepsilon)$ while the CPDA reaches the configuration $(bN_1 \dots N_l, S)$ representing $(bN_1 \dots N_l, \rho, \varepsilon)$. Therefore $KPTree(M)$ and $CTree(\mathcal{B}(unc(M))$ have root b and their respective subtrees are $KPTree(N_1, \rho, \varepsilon), \dots, KPTree(N_l, \rho, \varepsilon)$ and are $CTree(N_1, S), \dots, CTree(N_l, S)$. But since $(bN_1 \dots N_l, S)$ is representing $(bN_1 \dots N_l, \rho, \varepsilon)$, we get that for every i in $[1, l]$, (N_i, S) represents (N_i, ρ, ε). Repeating this argument *ad infinitum* we obtain that the trees $KPTree(M)$ and $CTree(\mathcal{B}(unc(M)))$ are identical. □

Example 9. Figure 7, we now illustrate this translation on the λY-term

$$M = \left(Y\,\mathbf{f}.\,\lambda g.\,g\,(\mathbf{f}(\lambda xy.\,gyx))\,(\mathbf{f}(\lambda xy.\,gyx))\right)(\lambda xy.x)$$

which is uncurried into

$$S = \left(Y\,\mathbf{f}.\,\lambda g.\,\pi_1(g)\,\langle \mathbf{f}(\lambda x.\,\pi_1(g)\langle \pi_2(x), \pi_1(x)\rangle),\, \mathbf{f}(\lambda x.\,\pi_1(g)\langle \pi_2(x), \pi_1(x)\rangle)\rangle\,\right)(\lambda x.\,\pi_1(x)).$$

So as to obtain the shorter representations of the machines we use the following shorthands:

$$P_1 = \lambda x.\, \pi_1(x), \qquad Q = \lambda x.\, \pi_1(g)\langle \pi_2(x),\, \pi_1(x)\rangle, \qquad T = \mathbf{f} Q,$$
$$G = \lambda g.\, \pi_1(g)\langle T, T\rangle, \qquad F = Y\mathbf{f}.\, G, \qquad S = FP_1\ .$$

The representations of configurations of CPDA and of Krivine machines are as before, but since now the stack can have at most one element, this element is represented by a box marked *arg*. The execution shows most of the cases related to the simulation of the Krivine machine by the CPDA: the first step is the replacement of the Y-variable $\mathbf{f}$ by its definition, next step eliminates Y binder, etc.

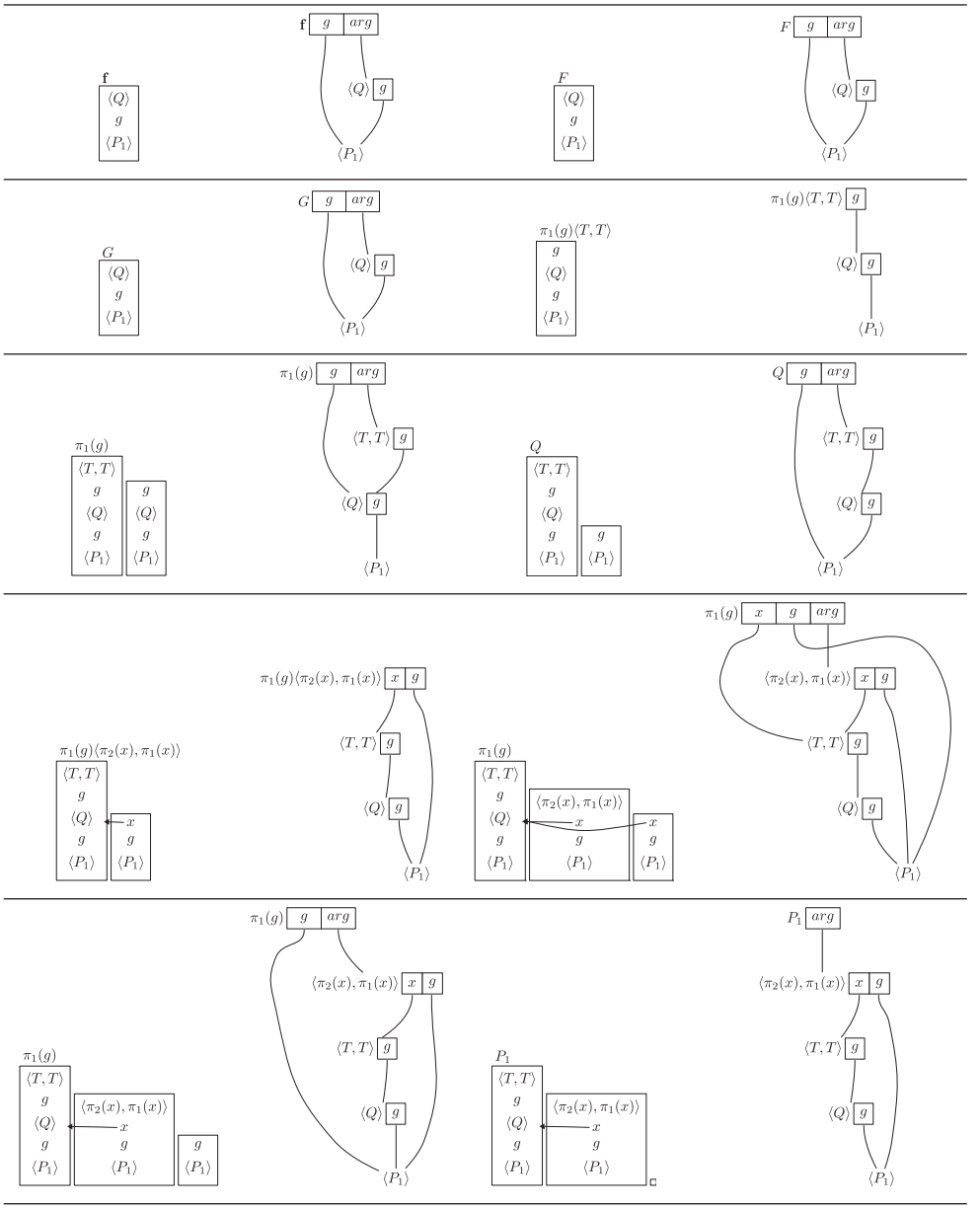
The safety condition in the context of η-long terms in canonical forms that are uncurried becomes rather simple: indeed, a term N which is not a tuple is safe when all its free λ-variables have order greater or equal to the order of N; a tuple $\overrightarrow{N_l}$ is safe when for every $1 \leqslant i \leqslant l$, N_i is safe. The reason of this simplification comes from the fact that in this context, we do not need to account for partially applied functions which is the only motivation for the notion of almost safe terms. Moreover, as mentioned in Blum and Ong (2009), safety is preserved by η-expansion and it can easily be seen that it is also preserved by uncurrying.

As for the previous translation (cf. Theorem 32), it can be shown that when one starts with a safe term M the CPDA from the above theorem only performs *collapse* operations that can be replaced by a single *pop* operation of an appropriate order. For this, it suffices to define a notion of *active links* of a configuration and show similarly in Lemma 31 that the transition rules of the CPDA preserve the fact that all active links have depth one when the terms used are safe. Because the collapse pointers are directly represented as the level of λ-variables, the proof is here very simple and can be omitted.

6. Comparison of various evaluations of higher-order programs with CPDA

The two translations we have proposed for evaluating λY-terms with CPDA differ in at least three ways:

1. The first translation uses stack variables to access arguments stored on the stack. The second one uses a different technique that consists in having at most one tuple of arguments, and annotating λ-variables with projections. This annotation tells which term in a tuple has to be evaluated. The first approach could be phrased as *autonomous arguments* while the second as *composite arguments*.
2. The first translation simply pushes on the stack all arguments, and duplicates the stack only when searching the value of a λ-variable. The second, duplicates the stack as soon as an argument is pushed on the stack so as to be closer to the Krivine machine representation of closures. The first approach could be described as *lazy copying*, while the second could be described as *eager copying*.
3. To find the value of a variable the first translation may perform an *a priori* unbounded number of collapse operations. In the second approach, one collapse is sufficient.



$$P_1 = \lambda x.\, \pi_1(x), \qquad Q = \lambda x.\, \pi_1(g)\langle \pi_2(x),\, \pi_1(x)\rangle, \qquad T = \mathbf{f}Q,$$
$$G = \lambda g.\, \pi_1(g)\langle T, T\rangle, \qquad F = Y\mathbf{f}.\, G, \qquad S = FP_1\ .$$

Fig. 7. Simulation of the Krivine machine with a CPDA.

The two approaches we have presented differ by three parameters that can all take two values. By choosing independently the value for those parameters, there might be six more CPDA evaluations of λY-terms. From this point of view the translation of Carayol and Serre (2012) can be seen as another approach using composite arguments. When evaluating at term $M = PN_1 \dots N_k$, where P is not an application (so in that context it is either a nonterminal or a λ-variable), they push M on the stack and evaluate P. Pushing M on the stack is done to store the tuple of the arguments of P. When P is a variable, say x, to evaluate P they systematically start with a $copy_{m_0-order(x)+1}$ operation where m_0 is the order of the scheme. So they use a lazy copying approach. As for the third parameter of our classification, in order to get the value associated to a parameter, their method may require the use of several *collapse* operations, but this number is bounded by the maximal arity of types occurring in the scheme. When compared to the second translation, which uses the *collapse* operation exactly once to retrieve an argument, it seems that the difference stems from the fact that the second translation uses terms in η-long form. It may thus be the case that the existence of a bound on the number of *collapse* operations needed to retrieve a variable depends on whether the encoding uses composite or autonomous arguments.

In the problems of model checking the trees generated by CPDAs, the number of states of the CPDA plays an important role when analysing the fixed-parameter tractability of the problem (Broadbent *et al.* 2012; Kobayashi 2011). In the translations we have provided the number of states is about the size of the λY-term the CPDA is simulating. This number can be reduced by discriminating two phases in the behaviours of the CPDAs obtained with the translations we have proposed. The first phase consists in decomposing terms into subterms while the second consists in searching for a variable value. While the states necessary can be made independent of the λY-term for the first phase by using some usual techniques to encode the state as stack symbols, it is not the case for the second phase where the names of the λ-variables (and the number of the stack variable for the first translation) need to be present in states. A method to reduce this number consists in finding an element of the α-equivalence class of the λY-terms that requires as few λ-variable names as possible. When this is done, the number of states of the CPDA obtained in our translations is linear in the maximal arity of the types occurring in the scheme that would represent the λY-term via the translation given in Section 3.2.

7. Conclusions

We have argued that the λY-calculus is a valuable framework for the analysis of recursive schemes. One could expect, that the translation theorems studied in this paper could be potentially much easier to prove for the restricted syntax of schemes. Yet, we show that the richer setting of λY-calculus offers many advantages, thanks to well-understood transformations on λ-terms like η-long normal form, or product types. The usefulness of clean evaluation mechanism in the form of the Krivine machine is hard to overestimate. Finally, at places where the syntax of schemes influences the results, we were able to resort to the canonical form of terms (the notion introduced in this paper).

Our motivation explains also why we have presented two translations. The first works for all λY-terms and requires only the notion of Krivine machine. We subsequently show that the translation behaves well in specialized cases: for terms in canonical form, and for safe terms. The proof of the correctness of the translation is short but is based on a non-obvious invariant. The second translation requires more definitions: it starts with terms in η-long form, then it uses product types, uncurrying, and the canonical form of terms. The gain is that the proof of correctness amounts to a rather straightforward verification. The two translations are also different in the way they simulate the evaluation of terms. The first translation delays all the modifications of the higher-order stack to the variable look-up. The second puts more constraints on the form of the stack, and in consequence requires also some stack manipulation in the case of the application rule.

The results presented here show that there is no inherent penalty when working with λY-calculus instead of recursive schemes. In consequence, it is possible to profit from the rich theory of the λY-calculus. An example of this approach is given in Salvati and Walukiewicz (2011). The paper presents the decidability of the model checking problem for λY-calculus and monadic-second order logic, and the proof heavily relies on Krivine machines. This result, in recursive schemes formulation, has been originally proved in Ong (2006) and then reproved in several different ways (Carayol *et al.* 2008; Kobayashi and Ong 2009). Recently, we have been able to solve a restricted version of the model checking problem using purely semantic means: models of λY-terms (Salvati and Walukiewicz 2013). As we show in op.cit. this model based approach allows, among others, to get powerful program transformation methods such as reflection property (Broadbent *et al.* 2010).

References

Aehlig, K., de Miranda, J. G. and Ong, C.-H. L. (2005) The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In: Proceedings of the International Conference on Typed Lambda Calculi and Applications. *Springer Lecture Notes in Computer Science* **3461** 39–54.

Aho, A. (1968) Indexed grammars – an extension of context-free grammars. *Journal of the Association of Computing Machinery* **15** (4) 647–671.

Barendregt, H. (1977) The type free lambda calculus. In: Barwise J.(ed.) *Handbook of Mathematical Logic*, Chapter D.7, North Holland, Amsterdam 1091–1132.

Barendregt, H. (1985) *The Lambda Calculus, Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics volume 103, North Holland, Amsterdam.

Barendregt, H. and Klop, J. W. (2009) Applications of infinitary lambda calculus. *Information and Computation* **207** (5) 559–582.

Blum, W. and Ong, L. (2009) The safe lambda calculus. *Logical Methods in Computer Science* **5** (1:3) 1–38.

Broadbent, C., Carayol, A., Ong, L. and Serre, O. (2010) Recursion schemes and logical reflection. In: *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society 120–129.

Broadbent, C. H., Carayol, A., Hague, M. and Serre, O. (2012) A saturation method for collapsible pushdown systems. In: Proceedings of the International Colloquium on Automata, Languages and Programming Track B. *Springer-Verlag Lecture Notes in Computer Science* **7392** 165–176.

Carayol, A., Hague, M., Meyer, A., Ong, C.-H. L. and Serre, O. (2008) Winning regions of higher-order pushdown games. In: *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society 193–204.

Carayol, A. and Serre, O. (2012) Collapsible pushdown automata and labeled recursion schemes equivalence, safety and effective selection. In: *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society 165–174.

Carayol, A. and Wöhrle, S. (2003) The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In: Foundations of Software Technology and Theoretical Computer Science. *Springer Lecture Notes in Computer Science* **2914** 112–124.

Caucal, D. (2002) On infinite terms having a decidable monadic theory. In: Proceedings of the International Symposium on Mathematical Foundations of Computer Science. *Springer Lecture Notes in Computer Science* **2420** 165–176.

Damm, W. (1982) The IO– and OI–hierarchies. *Theoretical Computer Science* **20** (2) 95–208.

Damm, W. and Goerdt, A. (1986) An automata-theoretical characterization of the OI-hierarchy. *Information and Control* **71** (1–2) 1–32.

Dezani-Ciancaglini, M., Giovannetti, E. and de' Liguoro, U. (1998) Intersection types, Lambda-models and Böhm trees. In: *MSJ-Memoir Vol. 2 'Theories of Types and Proofs'*, volume 2, Mathematical Society of Japan 45–97.

Hague, M., Murawski, A. S., Ong, C.-H. L. and Serre, O. (2008) Collapsible pushdown automata and recursion schemes. In: *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society 452–461.

Hyland, J. M. E. and Ong, C.-H. L. (2000) On full abstraction for pcf: I, II, and III. *Information and Computation* **163** (2) 285–408.

Ianov, Y. (1969) The logical schemes of algorithms. In: *Problems of Cybernetics I,* Oxford: Pergamon Press, 82–140.

Kfoury, A. and Urzyczyn, P. (1988) Finitely typed functional programs, Part II: Comparisons to imperative languages. *Technical report* BUCS tech report, 88-012, Boston University.

Knapik, T., Niwinski, D. and Urzyczyn, P. (2002) Higher-order pushdown trees are easy. In: Proceedings of the International Conference on Foundations of Software Science and Computation Structures. *Lecture Notes in Computer Science* **2303** 205–222.

Knapik, T., Niwinski, D., Urzyczyn, P. and Walukiewicz, I. (2005) Unsafe grammars and panic automata. In: Proceedings of the International Colloquium on Automata, Languages and Programming. *Springer Lecture Notes in Computer Science* **3580** 1450–1461.

Kobayashi, N. (2011) A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In: Proceedings of the International Conference on Foundations of Software Science and Computation Structures. *Springer Lecture Notes in Computer Science* **6604** 260–274.

Kobayashi, N. (2013) Model checking higher order programs. *Journal of the Association of Computing Machinery* **60** (3)Article No. 20. doi: 10.1145/2487241.2487246.

Kobayashi, N. and Ong, L. (2009) A type system equivalent to modal mu-calculus model checking of recursion schemes. In: *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society 179–188.

Krivine, J.-L. (2007) A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation* **20** (3) 199–207.

Maslov, A. (1974) The hierarchy of indexed languages of an arbitrary level. *Soviet Mathematics Doklady* **15** 1170–1174.

Maslov, A. (1976) Multilevel stack automata. *Problems of Information Transmission* **12** 38–42.

Milner, R. (1973) Models of LCF. *Memo AIM-186*, Stanford University.

Miranda, J. G. de (2006) *Structures Generated by Higher-Order Grammars and the Safety Constraint*, Ph.D. thesis, Oxford University.

Nivat, M. (1972a) Langages algébriques sur le magma libre et sémantique des schémas de programme. In: *Proceedings of the International Colloquium on Automata, Languages and Programming* 293–308.

Nivat, M. (1972b) On the interpretation of recursive program schemes. In: *Symposia Mathematica* **15** 255–281.

Ong, C.-H. L. (2006) On model-checking trees generated by higher-order recursion schemes. In: *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society 81–90.

Parys, P. (2012) On the significance of the collapse operation. In: *Proceedings of the Annual Symposium on Logic in Computer Science*, IEEE Computer Society 521–530.

Plotkin, G. D. (1977) LCF considered as a programming language. *Theoretical Computer Science* **5** (3) 223–255.

Salvati, S. and Walukiewicz, I. (2011) Krivine machines and higher-order schemes. In: Proceedings of the International Colloquium on Automata, Languages and Programming Track B. *Springer Lecture Notes in Computer Science* **6756** 162–173.

Salvati, S. and Walukiewicz, I. (2012) Recursive schemes, Krivine machines, and collapsible pushdown automata. In: Proceedings of the International Workshop on Reachability Problems. *Springer Lecture Notes in Computer Science* **7550** 6–20.

Salvati, S. and Walukiewicz, I. (2013) Using models to model-check recursive schemes. In: Proceedings of the International Conference on Typed Lambda Calculi and Applications. *Springer Lecture Notes in Computer Science* **7941** 189–204.