Recursive Schemes, Krivine Machines, and Collapsible Pushdown Automata

Sylvain Salvati and Igor Walukiewicz[⋆]

LaBRI, CNRS/Université Bordeaux, INRIA, France

Abstract. Higher-order recursive schemes offer an interesting method of approximating program semantics. The semantics of a scheme is an infinite tree labeled with built-in constants. This tree 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 the schemes do. We present two relatively simple translations from recursive schemes to CPDA using Krivine machines as an intermediate step. The later are general machines for describing computation of the weak head normal form in the lambda-calculus. They provide the notions of closure and environment that facilitate reasoning about computation.

1 Introduction

Recursive schemes are abstract forms of programs where the meaning of constants is not specified. In consequence, the meaning of a scheme is a potentially infinite tree labeled with constants obtained from the unfolding of the recursive definition. One can also see recursive schemes as another syntax for λY-calculus: simply typed λ-calculus with fixpoint combinators. In this context the tree generated by a scheme is called the Böhm tree of a term. Collapsible pushdown automata (CPDA) is another, more recent, model of the same generating power. In this paper we present two translations from recursive schemes and λY-calculus to CPDA. The translations use Krivine machines as an intermediate step.

Recursion schemes were originally proposed by Ianov as a canonical programming calculus for studying program transformation and control structures [12]. The study of recursion on higher types as a control structure for programming languages was started by Milner [21] and Plotkin [24]. Program schemes for higher-order recursion were introduced by Indermark [13]. Higher-order features allow for compact high-level programs. They have been present since the beginning of programming, and appear in modern programming languages like C++, Haskell, Javascript, Python, or Scala. Higher-order features allow to write code that is closer to specification, and in consequence to obtain a more reliable

⋆ Supported by ANR 2010 BLAN 0202 01 FREC

code. This is particularly useful in the context when 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[1].

Recursive schemes are an insightful intermediate step in giving a denotational semantics of a program. The meaning of a program can be obtained by taking the tree generated by the scheme and applying a homomorphism giving a meaning to each of the constants. Yet, in some cases the tree generated by the scheme 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 [17]. This is possible thanks to the fact that MSOL model checking is decidable for trees generated by higher-order recursive schemes [22].

The definition of the tree generated by the scheme, while straightforward, is somehow difficult to work with. Damm [9] has shown that considered as word generating devices, a class of schemes called safe is equi-expressive with higher-order indexed languages introduced by Aho and Maslov [2,19]. Those languages in turn have been known to be equivalent to higher-order pushdown automata of Maslov [20]. Later it has been shown that trees generated by higher-order safe schemes are the same as those generated by higher-order pushdown automata [15]. This gave rise to so called pushdown hierarchy [8] and its numerous characterizations [7]. The safety restriction has been tackled much more recently. First, because it has been somehow implicit in a work of Damm [9], and only brought on the front stage by Knapik, Niwiński, and Urzyczyn [15]. Secondly, because it required new insights in the nature of higher-order computation. Pushdown automata have been extended with so called panic operation [16,1]. This permitted to characterize trees generated by schemes of order two. Later this operation has been extended to all higher order stacks, and called collapse. Higher-order stack automata with collapse (CPDA) characterize recursive schemes at all orders [11]. The fundamental question whether collapse operation adds expressive power has been answered affirmatively only very recently by Parys: there is a tree generated by an order 2 scheme that cannot be generated by a higher-order stack automaton without collapse [23].

While recursive schemes and CPDA generated the same trees, they work in a very different way. It is relatively difficult to work directly with schemes since they do not provide any straightforward induction parameters. In contrast, induction on the stack size of CPDA has been successfully used in numerous instances. For example, the only approach known at present to prove, very useful, reflection theorem goes through CPDA.

In this paper we present two translations from recursive schemes to CPDA. The first translation of this kind for order 2 schemes has been done directly using the definition of a tree generated by the scheme [16]. It has been based on ideas of Kfoury and Urzyczyn from [14] where a similar translation but for call by value mode has been presented. At that time, this direct approach seemed too

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

cumbersome to generalize to higher orders. The first translation for schemes of all orders [11] used traversals, a concept based in game semantics. Very recently, Carayol and Serre [6] have presented a translation extending the one from [16] to all orders. This translation has been obtained independently form the one presented here. Indeed, the authors of [6] and the present paper have met in February 2011, and exchanged notes on the respective translations. The translation from [6] introduces already some notions of types in higher-order stack. We think that thanks to Krivine machine formulation our translations use even richer structure that allows for further substantial simplifications.

Our translation works on all λY-terms without a need to put them in a special form. Its drawback is that it may sometimes give a CPDA of order $m+1$ while m would be sufficient. We explain how to remove this drawback using some normalization of λY-terms. The second translation assumes that a given λY-term is in a special form, and it moreover uses product types. These preparatory steps allow for a translation where higher order stack reflects environments in a very direct way.

The structure of the paper is simple. In the next section we introduce the objects of our study: λY-calculus, schemes, Krivine machine, and collapsible pushdown automata (CPDA). We also present translations between 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 a λY-term in a canonical form. Looking at the restrictions on the syntax, 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. They both use Krivine machine as an intermediate step. The first works with all λY-terms, and is optimal with respect to order for terms in a canonical form. Hence in particular it is also optimal for recursive schemes. The second translation starts immediately from λY-terms in a canonical form, and is also optimal with respect to order. The two translations represent environments of configurations of Krivine machine in a very different way. The first delays all operations on the stack till a variable look-up is performed. The second is more direct, it starts with a fixed encoding of environments on the stack, and makes the structure of the stack always reflect this encoding.

2 Basic notions

In this preliminary section we introduce the basic objects of interest. We start with λY-calculus: a simply-typed lambda calculus with a fixpoint combinator. We use it as a more convenient syntax of recursive schemes. We briefly describe how schemes can be translated to λY-terms in a sense that the tree generated by a scheme is a Böhm tree of a term obtained from the translation (Lemma 2). We also show how λY-terms can be represented by schemes (Theorem 1). This 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 a canonical form. This form will be also very useful in the context of CPDA in later sections. Later in this section we present a more operational way of generating Böhm trees of terms, and here Krivine machines will come into the picture. Finally, we present collapsible pushdown automata and the trees they generate.

2.1 Simply typed lambda calculus and recursive schemes

Instead of introducing higher-order recursive schemes directly we prefer to start with the simply-typed lambda calculus with fixpoints: the *λY-calculus*. The two formalisms are essentially equivalent for the needs of this paper, but we opt for working with the later one. It gives us an explicit notion of reduction, and brings the classical notion of Böhm tree [3] that can be used directly to define the meaning of a scheme.

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

A *signature*, denoted Σ, is a set of typed constants, that is symbols with associated types from $\mathcal{T}$. We will assume that for every type $\alpha \in \mathcal{T}$ there is a constant ω^α standing for the undefined term of type α.

Of special interest to us will be *tree signatures* where all constants other than ω have order at most 2. Observe that types of order 2 have the form $0^i \to 0$ for some i.

The set of *simply-typed λ-terms* is defined inductively as follows. A constant of type α is a term of type α. For each type α there is a countable set of variables $x^\alpha, y^\alpha, \dots$ 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 $\alpha \to \beta$. If M is a term of type α and x^α is a variable of type α, then $Y x^\alpha.M$ is a term of type α. Finally, if M is of type $\alpha \to \beta$ and N is a term of type α then MN is a term of type β. The order of a term $\mathit{order}(M)$ is the order of its type. In the sequel we often omit the typing decoration of variables. For some technical convenience we does 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 λY-calculus.

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

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

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

We define the notion of *complexity* of a λ-terms to account for the complexity of its reductions. This notion will look at all subterms of a term but the constants ω.

Definition 1. *An* interesting subterm *of a term M, $isub(M)$, is a subterm other than constants ω. The complexity of a term M is m, when $m+1$ is the maximal order of an interesting subterm of M: $m+1 = \max\{order(N) \mid N \in isub(M)\}$. We write $comp(M)$ for the complexity of M.*

Notice that the order of any bound λ-variable x of a closed term M is smaller or equal to $comp(M)$.

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

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

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 λ-terms. It is customary in simply typed λ-calculus to work with terms in *η-long forms* that have a nice property of syntactically reflecting the structure of their typing. A term M is in *η-long* form when every 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. A particularity of closed terms in η-long form is that $comp(M)$ is equal to the maximal order of a λ-variable in M.

Thus, the operational semantics of the λY-calculus is the $\beta\delta$-reduction. It is well-known that this semantics 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 lambda calculus to consider a kind of infinite normal form that by itself is an infinite tree, and in consequence it is not a term of λY-calculus [3,10,5]. We define it below.

A *Böhm tree* is an unranked, ordered, and potentially infinite tree with nodes labeled by ω^α or terms of the form $\lambda x_1.\dots.x_n.N$ (we may write such a label $\lambda\overrightarrow{x}.N$); where N is a variable or a constant, and the sequence of lambda abstractions is optional. So for example x^0, $\lambda x.w^0$ are labels, but $\lambda y^0.x^{0\to 0}\ y^0$ is not.

Definition 2. *A* Böhm tree *of a term M is obtained in the following way.*

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

Observe that a term M has a $\beta\delta$-normal form if and only if $BT(M)$ is a finite tree without ω constants. 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 at the exception of Y and ω are of order at most 2. A closed term without λ-abstraction and Y 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 1. *If M is a closed term of type 0 over a tree signature then $BT(M)$ is a potentially infinite tree whose leaves are labeled with constants of type 0 and whose internal nodes with k children are labeled with constants of type $0^k \to 0$.*

Higher-order recursive schemes use a somehow simpler syntax: the fixpoint operators are implicit and so is the lambda-abstraction. A recursive scheme over a finite set of nonterminals $\mathcal{N}$ 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 variables and constants, other than Y and ω, using the application construction. Let us fix a tree signature Σ, and a finite set of typed nonterminals $\mathcal{N}$. A higher-order recursive scheme is a function $\mathcal{R}$ assigning to every nonterminal $F \in \mathcal{N}$, a term $\lambda\overrightarrow{x}.M_F$ where: (i) M_F is an applicative term, (ii) the type of $\lambda\overrightarrow{x}.M_F$ is the same as the type of F, and (iii) the free variables of M are among $\overrightarrow{x}$ and $\mathcal{N}$.

Definition 3. *The* order of a scheme *$\mathcal{R}$ is m if $m+1$ 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. It is a scheme of order 2.

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

2.2 From recursive schemes to λY-calculus

The translation from a recursive scheme to a lambda-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 nonterminals $\mathcal{N} = \{F_1, \ldots, F_n\}$. The term T_n representing the meaning of the nonterminal F_n is obtained as follows:

$$\begin{aligned} T_1 &= YF_1.\mathcal{R}(F_1) \\ T_2 &= YF_2.\mathcal{R}(F_2)[T_1/F_1] \\ &\ \vdots \\ T_n &= YF_n.(\ldots((\mathcal{R}(F_n)[T_1/F_1])[T_2/F_2])\ldots)[T_{n-1}/F_{n-1}] \end{aligned} \tag{1}$$

The translation (1) applied to the recursion scheme for *map* gives a term:

$$Y map^{(0\to0)\to0\to0}.\lambda f^{0\to0}.\lambda l^0.$$
$$\mathbf{if}\ (l = nil)\ nil\ \big(\mathbf{cons}\ (f(\mathbf{head}(l)))\ (map\ f\ (\mathbf{tail}(l))\big)$$

This time we have used the λ-calculus way of parenthesizing expressions.

We will not recall here a rather lengthy definition of a tree generated by a recursive scheme referring the reader to [15,9]. For us it will be sufficient to say that it is the Böhm tree of a term obtained from the above translation. For completeness we state the equivalence property. Anticipating contexts where the order of a scheme or a term is important let us observe that strictly speaking the complexity of the term obtained form the translation is bigger than the order of the schema.

Lemma 2. *Let $\mathcal{R}$ be a recursion scheme and let F_n be one of its nonterminals. A term T_n obtained by the translation* (1) *is such that $BT(T_n)$ is the tree generated by the scheme from nonterminal F_n. Moreover $comp(T_n) = order(\mathcal{R}) + 1$.*

2.3 From λY-calculus to recursive schemes

Of course, λY-terms can also be translated to recursive schemes so that the tree generated by the obtained scheme is the Böhm tree of the initial term. We are going to present two translations. The first one will be rather straightforward and will not assume any particular property of λY-terms it transforms. However the obtained transformation will not be dual to the one from Lemma 2 above in the sense that applying the two transformations one after another can increase the order of a scheme by 1. To obtain a dual transformation we will first need to transform a λY-term into a form that we call *canonical*. As we will see later, the transformation from Lemma 2 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 2 and back to scheme does not increase the order (Theorem 1).

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 important 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 $\mathcal{N}_M = \{\langle N\rangle \mid N \in isub(M)\}$; 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 $\alpha_1 \to \dots \to \alpha_n \to \alpha$. Each element $\langle N\rangle$ of $\mathcal{N}_M$ determines a single equation in $\mathcal{R}_M$, this equation is obtained following this table:

N	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 following lemma summarizes the properties of the translation. We omit the proof that follows directly from the definitions.

Lemma 3. *For every term M the tree generated by $\mathcal{R}_M$ defined by the table above is identical to $BT(M)$. The order of $\mathcal{R}_M$ is $comp(M)$.*

The order of the obtained scheme does not correspond to the order of the transformation 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 is recursive variables, in schemes have a special treatment. In order to obtain 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. For this purpose we consider that the set of variables is partitioned into 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 variable (both λ-variables and Y-variables) in a term M is bound at most once. This means that we can write $term_M(\mathbf{x})$ (*resp.* $term_M(x)$) for the unique subterm $Y\mathbf{x}.N$ (*resp.* $\lambda x.N$) of M starting with the binder of $\mathbf{x}$ (*resp.* x). We will omit a subscript M if it is clear from the context.

Definition 4. *The term M is in* canonical form *when it satisfies the following three 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 free variables in N are Y-variables.*

It is worth noticing that the transformation from schemes to terms we have described in the previous subsection produces terms in canonical form. We first remark an interesting property β-normal terms.

Lemma 4. *Given M a term of type 0 is in β-normal form, 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. As M is in β-normal form we have two cases:

- Variable, z appears in the sequence $\overrightarrow{z}$ in a term $Y\mathbf{x}.(\lambda\overrightarrow{z}.Q)$. Here we get immediately that $\mathit{order}(z) < \mathit{order}(\mathbf{x})$ 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 2 so the sequences $\overrightarrow{z_1}, \ldots, \overrightarrow{z_p}$ would be empty. In case $P = x$, we have $\mathit{order}(z) < \mathit{order}(x)$. But the λ-variable x is bound at a lower depth than z and by induction there is a Y-variable $\mathbf{x}$ such that $\mathit{order}(x) < \mathit{order}(\mathbf{x})$ and therefore $\mathit{order}(z) < \mathit{order}(\mathbf{x})$ which gives the expected result. In both the cases where $P = \mathbf{x}$ and where $P = \mathbf{x}$ we obviously we have $\mathit{order}(z) < \mathit{order}(\mathbf{x})$.

□

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

Lemma 5. *If M is a term of type 0, then there is a term M' in canonical form such that $BT(M') = BT(M)$ and $\mathit{comp}(M') \leq \mathit{comp}(M)$.*

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 lambda 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 $\mathit{comp}(M_1) \leq \mathit{comp}(M_0)$. Moreover M_1 satisfies the first condition of being in a canonical form (cf. Definition 4).

The second step involves removing non-recursive variables from fixpoint subterms. We replace every subterm $Y\mathbf{x}^\alpha.P$ of M_2 by $Q\overrightarrow{y}$ where

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

$\overrightarrow{y} = y_1^{\beta_1}\ldots y_m^{\beta_m}$ is a sequence of λ-variables free in P, and $\gamma = \beta_1 \to \cdots \to \beta_m \to \alpha$. Since, by Lemma 4, the orders of $\beta_1, \ldots, \beta_m$ are strictly smaller than $\mathit{comp}(M_1)$, since, moreover, $\mathit{order}(\alpha) \leq \mathit{comp}(M_1)$, we have that $\mathit{order}(\gamma) \leq \mathit{comp}(M_1)$. This transformation thus gives a term M_2 such that $\mathit{comp}(M_2) = \mathit{comp}(M_1)$. 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 with respect to putting into a context (see [4]), we will get desired $BT(M_2) = BT(M_1)$. To see that $BT(Q\overrightarrow{y}) = BT(Y\mathbf{x}^\alpha.P)$, let us first remark that:

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

By iterating this sequence of reduction steps we obtain

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

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 $comp(M_2) \leq comp(M_0)$. $\square$

It remains to present a transformation of term M in a canonical form into a scheme $\mathcal{R}_M$ such that $order(\mathcal{R}_M)+1 = comp(M)$. This is made possible thanks to the distinction between the two kinds of variables. The scheme obtained by 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 an important subterm N of M, we let $nr(N)$ 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 a subterm of M. If N has a type α and $[N]$ is in $\mathcal{N}_M$, then its type will be $\alpha_1 \to \dots \to \alpha_n \to \alpha$ when $nr(N) = x_1^{\alpha_1}\dots x_n^{\alpha_n}$. We associate with each element $[N]$ of $\mathcal{N}_M$ an equation according to the following table:

N	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
N_1N_2	$[N_1N_2] \equiv \lambda\overrightarrow{x}.[N_1]\overrightarrow{y}([N_2]\overrightarrow{z})$ where $nr(N_1N_2) = \overrightarrow{x}$, $nr(N_1) = \overrightarrow{y}$, and $nr(N_2) = \overrightarrow{z}$
$\lambda y.P$	$[\lambda y.P] \equiv \lambda\overrightarrow{x}y.[P]\overrightarrow{z}$ where $y \in NV$, $nr(\lambda y.P) = \overrightarrow{x}$, and $nr(P) = \overrightarrow{z}$
$Y\mathbf{x}.P$	$[Y\mathbf{x}.P] \equiv [P]$

Let us compare this translation to the previous one. First, Y-variables are not translated at all. The cases for application and abstraction are 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 simply because recursive terms do not contain free λ-variable.

Lemma 6. *For every λY-term M of type 0 in canonical form the scheme $\mathcal{R}_M$ defined by the table above generates the tree $BT(M)$, and moreover $order(\mathcal{R}) + 1 = comp(M)$.*

Proof. It is easy to check that the tree generated by $\mathcal{R}_M$ is $BT(M)$. We are going to explain why $order(\mathcal{R}) + 1 = comp(M)$. Recall that $order(\mathcal{R}) + 1$ is the maximal order of a nonterminal in $\mathcal{R}$. Similarly $comp(M) + 1$ is equal to $\max\{order(N) \mid N \in isub(M)\}$.

Let $m = comp(M)$. By definition, if $\mathbf{x}$ is a Y-variable from M then $order(\mathbf{x}) \leq m$. From Lemma 4 we get that if y is a λ-variable in M then $order(y) < m$. It then follows that the order of a nonterminal $[\lambda y.P]$, for a non-recursive variable y, is at most m. The order of a nonterminal $[Y\mathbf{x}.P]$, is at most m too. This shows that the order of every nonterminal in $\mathcal{R}$ is bounded by m. Hence $order(\mathcal{R}) = m-1$ as required. □

Combining Lemma 6 and Lemma 5 we obtain:

Theorem 1. *For every closed term M of type 0, there is a scheme $\mathcal{R}$ generating the tree $BT(M)$ and such that $order(\mathcal{R})+1 \leq comp(M)$.*

2.4 Krivine machines

Lazy evaluation with static scope

A Krivine machine [18], 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:

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

As in this grammar, we will use $\emptyset$ for the empty environment. The notation $\rho[x \mapsto C]$ represent 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 lambda term denoted by the closure $\rho(x)$.

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:

Small-steps operational semantics

$$\begin{aligned}
(\lambda x.M,\rho,(N,\rho')S) &\to (M,\rho[x\mapsto (N,\rho')],S)\\
(MN,\rho,S) &\to (M,\rho,(N,\rho)S)\\
(Yx.M,\rho,S) &\to (M,\rho[x\mapsto (Yx.M,\rho)],S)\\
(x,\rho,S) &\to (M,\rho',S) \qquad \text{where } (M,\rho')=\rho(x)
\end{aligned}$$

Note that the machine is deterministic. We will write $(M,\rho,S) \to^* (M',\rho',S')$ to say that 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 put the argument N on the stack together with the current closure that permits to evaluate N when needed; 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 calculate M.

Finally, the rule for 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 also an environment giving the right meanings of free variables in the term.

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

Lemma 7. *If M is a simply typed term of type 0 and (N, ρ, S) is a configuration reachable from the initial configuration $(M, \emptyset, \varepsilon)$ then*

- *N is a subterm of M, hence it is simply typable.*
- *Environment ρ associates to a variable x^α a closure (K, ρ') so that K has type α; we will say that the closure is of type α too. Moreover K is a subterm of M.*
- *The number of elements in S is determined by the type of N: there are k elements when the type of N is $\alpha_1 \to \cdots \to \alpha_k \to 0$.*

Let us explain how to use Krivine machines to calculate the Böhm tree of a term. 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 $\mathit{KTree}(M, \rho, \varepsilon)$ be the tree consisting only of a root labeled with ω 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 different from ω and Y. In this situation $\mathit{KTree}(M, \rho, \varepsilon)$ has b in the root and for every $i = 1, \dots, k$ it has a subtree $\mathit{KTree}(N_i, \rho_i, \varepsilon)$. Due to typing invariants we have that k is the arity of the constant b. Since we are working with tree signature, b has order at most 2, and in consequence all terms N_i have type 0.

Definition 5. *For a closed term M of type 0 we let $\mathit{KTree}(M)$ be $\mathit{KTree}(M, \emptyset, \varepsilon)$; where $\emptyset$ is the empty environment, and ε is the empty stack.*

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

Lemma 8. *For every closed term M of type 0 over a tree signature: $\mathit{KTree}(M) = \mathit{BT}(M)$.*

2.5 Collapsible pushdown automata

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 collapsible pushdown automata, and explain how they can be used to generate infinite trees. In this subsection we fix a tree signature Σ.

A stack of order m is a stack of stacks of order $m - 1$. Let Γ be a stack alphabet. *Order* 0 *stack* is a symbol from Γ. *Order m stack* is a nonempty 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 a m-stack S, $top(S)$, is S if $m = 0$, and the topmost element of the topmost $(m-1)$-stack of S otherwise.

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.

The operations on a stack of order m are indexed by their order $i \in \{1, \dots, m\}$ when needed. We have pop_i, $copy_i$, $push_1^{a,i}$ for $a \in \Sigma$, and *collapse*. On a stack $S = [S_1 \dots S_{l+1}]$ of order $j \geq i$ these operations are defined as follows:

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

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

Here $S_l^{\upharpoonright i}$ is S_l with all the superscripts (i, k_i), for some k_i, replaced by $(i, k_i + 1)$.

$$push_1^{a,i}(S) = \begin{cases} [S_1 \dots S_l S_{l+1} a^{i,1}] & \text{if } Ord(S) = 1 \\ [S_1 \dots S_l\ push^{a,i}(S_{l+1})] & \text{otherwise} \end{cases}$$

So we can push new elements only on the topmost order 1 stack: stacks of bigger order are created with *copy* operation. 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.

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

The collapse operation performs 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:

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

The idea is that a state and a top stack symbol 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 $\perp \in \Gamma$ used to denote the bottom of the stack. We will also denote by $\perp$ the initial stack containing only $\perp$.

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 the tree consisting of only a root labeled ω 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 $\delta(q', top(S')) = (b, q_1, \dots, q_k)$ for some constant b. In this situation $CTree(q, S)$ has the root b and for every $i = 1, \dots, k$ it has $CTree(q_i, S')$ as a subtree.

Definition 6. *For a CPDA $\mathcal{A}$ we let $CTree(\mathcal{A})$ be $CTree(q^0, \perp)$; where q_0 is the initial state of $\mathcal{A}$, and $\perp$ is the initial stack.*

3 From λY-calculus to CPDA

As we have seen, recursive schemes can be translated to λY-terms,and vice-versa (cf. Sections 2.2 and 2.3). In this section we will show how, for a λY-term M_0, to construct a CPDA $\mathcal{A}$ such that the tree generated by $\mathcal{A}$, that is $CTree(\mathcal{A})$, is $BT(M_0)$. For this we will use the characterization of $BT(M_0)$ in terms of Krivine machine.

The first step will be to represent differently configurations of the Krivine machine. This is done purely for reasons of exposition. Then we will present the construction of $\mathcal{A}$ 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 4). The order of the stack of $\mathcal{A}$ will be the same as the order of arguments of M_0. Put together with the translation from Lemma 2 this does not give an optimal, with respect to order, translation from recursive schemes to CPDA. In the last subsection we explain how to avoid this problem using some simple manipulations on λY-terms and Krivine machines.

For the entire section we fix a tree signature Σ.

3.1 Stackless Krivine machines

From Lemma 7 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.

We will use a variable γ_i to represent the i-th 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, 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. The new rules of the Krivine machine become

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

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 of 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.

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

- $M'=M$.
- for every normal variable x: $\rho(x)=\rho'(x)$,
- k is the number of elements on the stack S and $\rho'(\gamma_i)$ is the i-th element on S for $i=1,\dots,k$; 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 9. *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 we can use stackless Krivine machines in constructing $KTree(M)$ (cf. Definition 5) which is no other than $BT(M)$.

3.2 Simulation

Fix a closed term M_0, let m be the complexity of M_0 (cf. Definition 1). We want to simulate the computation of the stackless Krivine machine on M_0 by a CPDA with collapse with stacks of order m.

The idea is that a configuration (M,ρ,k) will be represented by a state (M,k) of the machine and the higher order stack encoding ρ. 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

$$(x,\gamma_i),(x,N),\ (\gamma_i,N),\ (\gamma_i,\bot),\quad \text{and}\quad sp_l \text{ for } l=1\dots,m.$$

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 denoting stack pointer as it will be explained below. 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 element. 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:

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

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

The first function traverses the top-most order 1 stack looking for a pair determining the value of the variable z. It will be always the case that this value is a stack variable and then the second function is called to get to the real value of the variable. 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 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 *find* function continues the search on the top-most 1-stack.

With the help of these two functions, the environment $\rho[S]$ determined by S is defined as follows:

Definition 7. *A stack S determines an environment $\rho[S]$ as follows:*

$$\begin{aligned} \rho[S](x) &= (N, \rho[S']) \quad \textit{if } (N, S') = value(x, S) \textit{ and } x \textit{ normal variable} \\ \rho[S](\gamma_i) &= (N, \rho[S']) \quad \textit{if } (N, S') = find(\gamma_i, S) \textit{ and } \gamma_i \textit{ stack variable} \end{aligned}$$

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

The following simple observation is the central place where the *collapse* operation is used. Since the *value* 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 10. *For every $d = 2, \dots, m$: if $S' = copy_d(S)$ then $\rho[S] = \rho[S']$*

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

Definition 8. *Let M_0 be a closed term M_0 and m_0 be the maximal order of a variable appearing in M_0. We define m_0-CPDA $\mathcal{A}(M_0)$ whose states are pairs (M, k), where M is a subterm of M_0 and k a number of its arguments. 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 a state (z, k), with z a variable we compute the number called the link order of the variable: $ll(z) = m_0 - order(z) + 2$.*
  If $ll(z) \leq m$, the automaton does
  $$(N, S') = value(z, copy_{ll(z)}(S)), \quad and \quad S'' = push_1^{sp_{order(z)},ll(z)}(S').$$
  If $ll(z) = m + 1$ then the automaton just does
  $$(N, S') = value(z, S) \quad and \quad S'' = S'.$$
  The new state is (N, k) and the new stack is S''. These operations implement the search for a value of the variable inside the higher-order stack. The copy operation is necessary to preserve arguments of z. In the special case when $ll(z) = m_0 + 1$, variable z has type 0 so it has no arguments and we do not need to do a copy operation.
– *In a state $(\lambda x.M, k)$ the automaton does*
  $$S' = push_1^{(x,\gamma_k),1}(S) \quad and \quad S'' = push_1^{(\gamma_k,\perp),1}(S')$$
  The new state is $(M, k-1)$ and the new stack is S''. These two operations implement 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) the automaton does*
  $$S' = push_1^{(\gamma_{k+1},N),1}(S)$$
  the state becomes $(M, k+1)$ and the stack S'. So the variable γ_{k+1} gets assigned $(N, \rho[S])$.
– *In a state $(Yx.M, k)$ the automaton does*
  $$S' = push_1^{(x,Yx.M),1}(S)$$
  the state becomes (M, k) and the stack S'.
– *In a state (b, k) with b a constant from Σ of arity k the automaton goes to $(b, qf_1, \ldots, qf_k)$. 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 an going to its arguments.*

Let us comment on this definition. The case of (z, k) is the most complicated. Observe that if $order(z) = m_0$ then $ll(z) = 2$, and if $order(z) = 1$ 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 $S_1 = \mathit{copy}_{ll(z)}(S)$ followed by $S_2 = \mathit{push}_1^{a,ll(z)}(S_1)$ and $S_3 = \mathit{collapse}(S_2)$ then $S_3 = S$; in other words we recover the original stack. We will prove later that the *value* operation destroys only the part of the stack of order $< ll(z)$.

Observe that apart from the variable case, the automaton uses just the push_1 operation.

For the proof of correctness we will need one more definition. We define argument order of a higher-order stack S to be the maximal order of types of elements assigned to stack variables.

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

We are going to show that the CPDA defined above simulates the computation of the Krivine machine step by step. For the proof we need to formulate an additional property of a stack S:

> $(*)$ For every element sp_l in S: (i) the collapse pointer at sp_l is of order $d = m_0 - l + 2$, and (ii) $l > ao(\mathit{collapse}(S'))$ 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. In other words, the orders of these stack variables give an upper bound on the collapse order d since d depends inversely on l. This is a very important property as it will ensure that we will not destroy a sack too much when we look for the value of a variable.

We are ready to prove that the CPDA simulates Krivine machine. It is the only technical lemma needed to prove the correctness of the translation.

Lemma 11. *Let $(M, \rho, k) \to (M_1, \rho_1, k_1)$ be two successive configurations of the stackless Krivine machine. Let S be a higher order stack satisfying the condition $(*)$ and $\rho[S] = \rho$. From the state (M, k) and the stack S the CPDA $\mathcal{A}(M_0)$ reaches the state (M_1, k_1) with the stack S_1 satisfying $(*)$ condition and $\rho_1 = \rho_1[S_1]$.*

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

$$(z, \rho, k) \to (N, \rho'', k) \quad \text{where } (N, \rho') = \rho(z)$$
$$\text{and } \rho'' = \rho'[\gamma_1/\rho(\gamma_1), \dots \gamma_k/\rho(\gamma_k)]$$

Let us first examine the simpler case when $\mathit{order}(z) = 1$. In this case $k = 0$. By hypothesis the CPDA is in the state $(z, 0)$ and $\rho = \rho[S]$. We have that the CPDA does $(N, S') = \mathit{value}(z, S)$, the new state becomes $(N, 0)$ and the new stack is S'. Since $\rho(z) = \rho[S](z)$, by the definition of the later we have $\rho[S'] = \rho'$, and we are done.

Now consider the case when $order(z) > 1$. By hypothesis the CPDA is in the state (z, k) and $\rho = \rho[S]$. We recall the operation of the stack machines that are performed in this case:

$$(N, S') = value(z, copy_{ll(z)}(S)), \quad \text{and} \quad S'' = push_1^{sp_{order(z)}, ll(z)}(S').$$

By Lemma 10 and the assumption of the 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 $\rho[S'']$.

It remains 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 and the meaning of *find* we have $\mathit{find}(\gamma_i, S'') = \mathit{find}(\gamma_i, collapse(S''))$. It is enough to show that $collapse(S'') = S$.

Recall that $(N, S') = value(z, copy_{ll(z)}(S))$. Let us examine the behaviour of *value* function. We will show that it only destroys a part of the stack that has been put on top of S with the $copy_{ll(z)}$ operation. First, the *value* function does a sequence of pop_1 operations until it gets to a pair (z, γ_i) for some γ_i. We have that $order(\gamma_i) = order(z)$. The operation *find* is then started. This operation does pop_1 operations 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 $l > ao(S_1) \geq order(z)$. By the invariant $(*)$ the collapse done on S_1 is of order $m_0 - l + 2 < m_0 - order(z) + 2 = ll(z)$. Repeating this argument we see that during the *value* operation the only stack operations are pop_1 and *collapse* of order smaller than $ll(z)$. This implies that the *value* operation changes only the topmost $ll(z)$ stack. So $collapse(S'') = S$ as required.

It remains to show that condition $(*)$ holds. The first part of the condition follows from the definition as $ll(z) = m_0 - order(z) + 2$. The second part of the condition is clearly satisfied by S' since it is preserved by the *copy* operation. Next we do $S'' = push^{sp_{order}(z), ll(z)}(S')$. By the above, we have that $pop_{ll(z)}(S') = S$. Now since the elements on the stack are the arguments of z we have that for all i it holds that $order(z) > order(N_i)$ where $(N_i, \rho_i) = \rho(\gamma_i)$. Since $\rho = \rho[S]$ we have $order(z) > ao(S)$. This shows the second condition. □

Theorem 2. *Consider terms and automata over a tree signature Σ. For every term M of type 0 there is a CPDA $\mathcal{A}$ such that $BT(M) = CTree(\mathcal{A})$ The order of $\mathcal{A}$ is the same as the maximal order of a variable appearing in M.*

Proof. Using Lemma 1 we consider $KTree(M)$ instead of $BT(M)$. The tree $KTree(M)$ starts with the execution of a Krivine machine from $(M, \emptyset, \varepsilon)$. By Lemma 9, we can as well look at the execution of the stackless Krivine machine from $(M, \emptyset, 0)$. We take automaton $\mathcal{A}(M)$ as defined above. It is m CPDA by definition.

The $CTree(\mathcal{A}(M))$ starts from the configuration consisting of the state $(M, 0)$ and stack $\perp$. It is clear that $\rho[\perp] = \emptyset$ and $\perp$ satisfies $(*)$ condition. Repeated applications of Lemma 11 give us that either both trees consist of the root labeled 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 $(*)$. By definitions of both trees, they will have b in the root and this root will have k subtrees. The i-th subtree on the one side will be $\mathit{KTree}(N_i,\rho_i,0)$ where $(N_i,\rho_i)=\rho(\gamma_i)$. On the other side it will be $\mathit{CTree}((N_i,0),S_i)$ where $(N_i,S_i)=\mathit{find}(\gamma_i,S)$. We have by definition of $\rho[S]$ that $\rho[S_i]=\rho_i$. Since S_i is a initial part of the stack S it satisfies $(*)$ condition too. Repeating this argument ad infinimum we obtain that the trees $\mathit{KTree}(M)$ and $\mathit{CTree}(\mathcal{A}(M))$ are identical. $\square$

3.3 Lowering the order

If we start from a recursive scheme of order m, the translation from Lemma 2 will give us a term with complexity $m+1$ (cf. Definition 1). So the construction from Theorem 4 will produce a CPDA working with stack of order $m+1$. Nevertheless, it is possible to produce an m-CPDA. For this, we recall that a term obtained from a translation of a recursive scheme is in a canonical form (Definition 4). 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 boldface font. As we show below, this form allows to store only λ-variables in the environment. Since by Lemma 4 they have smaller order than the Y-variables, we get the desired optimization.

We assume that the is bound variable in the term have are pairwise distinct. Recall also that we use $\mathit{term}(\mathbf{x})$ to denote the sub-term starting with the binder of $\mathbf{x}$. We add two new rules to the stackless Krivine machine to cater for recursive variables:

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

The first rule replaces the original fixpoint rule: the difference is that the value of the Y-variable $\mathbf{x}$ is not stored on the stack. The second rule tells what to do when we encounter Y-variable; this is needed since $\mathbf{x}$ will not appear on the environment. It is not difficult to see that on terms in canonical form the two rules faithfully simulate the original Krivine machine: we simply store only λ-variables in the environment.

It is easy to adapt the construction of a CPDA from a term (Definition 8) 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 $(\mathit{term}(\mathbf{x}),k)$. It is straightforward to check that for the modified Krivine machine and the modified CPDA Lemma 11 still holds.

As M is in the canonical form and has complexity $m+1$, by Lemma 4 the only variables that are of order $m+1$ are recursive. Thanks to modified translation CPDA stores on its stack only λ-variables, and these have order at most m.

Hence in the Definition 8 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.

Theorem 3. *Consider terms and automata over a tree signature Σ. For every term M of type 0 there is a CPDA $\mathcal{A}$ such that $BT(M) = CTree(\mathcal{A})$. If M is in the canonical form then the order of $\mathcal{A}$ is the same as the maximal order of a λ-variable appearing in M. In particular, if M is obtained by the translation (cf. Theorem 1) of a recursive scheme of order m then CPDA is also of order m.*

4 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. Instead of incorporating the stack in its environment, it makes the stack structure simpler by keeping it either empty or a singleton. The translation also differs in the way closures are represented in the CPDA and the way the closures associated to variables on the higher-order stack of the CPDA are retrieved. This translation is a bit more technical to define but it is then slightly easier to prove its correctness.

4.1 Uncurrification of λ-terms and Krivine machines with product

For this translation, we need to start with a term in η-long form and in the canonical form (see Section 2.1 and Definition 4). We assume that variables are partitioned between λ-variables and Y-variables and as we did before, we shall note the latter in boldface font. We then apply a syntactic transformation to the term called *uncurryfication*. 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 counterpart 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 by the rule: $\pi_i(\langle M_1, \ldots, M_n\rangle) \to_\beta M_i$. As a short hand 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: $\mathit{order}(0) = 1$ and $\mathit{order}(\gamma \to \alpha) = \max(\mathit{order}(\gamma) + 1, \alpha)$ and $\mathit{order}(\alpha_1 \times \cdots \times \alpha_n) = \max_{i\in[1;n]}(\mathit{order}(\alpha_i))$. The order of a term is the order of its type.

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

1. $\mathit{unc}(0) = 0$
2. $\mathit{unc}(\alpha_1 \to \cdots \to \alpha_n \to 0) = (\mathit{unc}(\alpha_1) \times \cdots \times \mathit{unc}(\alpha_n)) \to 0$,
3. $\mathit{unc}((Y\mathbf{x}^\alpha.P)N_1 \dots N_m) = (Y\mathbf{x}^{\mathit{unc}(\alpha)}.\mathit{unc}(P[\mathbf{x}^{\mathit{unc}(\alpha)}/\mathbf{x}^\alpha]))T$ in case when $T = \langle \mathit{unc}(N_1), \dots, \mathit{unc}(N_m)\rangle$,
4. $\mathit{unc}(\lambda x_1^{\alpha_1} \dots x_n^{\alpha_n}.P) = \lambda z^\gamma.\mathit{unc}(P[\pi_1(z^\gamma)/x_1^{\alpha_1}, \dots \pi_n(z^\gamma)/x_n^{\alpha_n}])$ where P has an atomic type and $\gamma = \mathit{unc}(\alpha_1) \times \cdots \times \mathit{unc}(\alpha_n)$,
5. $\mathit{unc}(\pi_i(z^\gamma)N_1 \dots N_m) = \pi_i(z^\gamma)\langle \mathit{unc}(N_1), \dots, \mathit{unc}(N_m)\rangle$,
6. $\mathit{unc}(\mathbf{x}^\alpha N_1 \dots N_m) = \mathbf{x}^\alpha \langle \mathit{unc}(N_1), \dots, \mathit{unc}(N_m)\rangle$,
7. $\mathit{unc}(aN_1 \dots N_m) = a\, \mathit{unc}(N_1) \dots \mathit{unc}(N_m)$.

Notice that for an uncurrified type, $\gamma \to 0$, we have $\mathit{order}(\gamma \to 0) = \mathit{order}(\gamma)+1$. As we have already mentioned, the role of the uncurrification is to make the stack of the Krivine machine simple. Indeed with an uncurrified 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 by 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 the Böhm tree of the uncurrified form of a term is the same as the Böhm tree of the original term.

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

We now give an adaptation of the Krivine machine that computes the Böhm tree of a term like M. For this, we slightly change the notion of environment. In the definition of the Krivine machine in section 2.4, we considered that environments were functions from variables to closures. Here, so as to emphasize the way we make CPDA retrieve the closure associated to variable, we structure the environment of the Krivine machine as an association lists 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:

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

where :: denotes the operation of appending an element on top of a list. As a short hand, 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:

$$\begin{array}{rcl}
(M\overrightarrow{N_p}, \rho, \varepsilon) & \to & (M, \rho, (\overrightarrow{N_p}, \rho)) \\
(\lambda x.M, \sigma, (\overrightarrow{N_p}, \rho')) & \to & (M, (x, \overrightarrow{N_p}, \rho') :: \rho, \varepsilon) \\
(\pi_i(x), (x, \overrightarrow{N_p}, \rho') :: \rho, \sigma) & \to & (N_i, \rho', \sigma) \\
(\pi_i(x), (y, \overrightarrow{N_p}, \rho') :: \rho, \sigma) & \to & (\pi_i(x), \rho, \sigma) \text{ when } x \neq y \\
(\mathbf{x}, \rho, \sigma) & \to & (\mathit{term}(\mathbf{x}), \rho, \sigma) \\
((Y\mathbf{x}.P), \rho, \sigma) & \to & (P, \rho, \sigma)
\end{array}$$

So as to make precise the relationship between this version of the Krivine machine computing the Böhm tree of uncurrified terms, and the Krivine machine we have defined section 2.4, we define the notion of $\mathit{KPTree}(M, \rho, \sigma)$ of the Böhm tree computed by the Krivine machine on uncurrified terms from the configuration (M, ρ, σ). If from the configuration (M, ρ, σ) the machine reaches a configuration $(b\overrightarrow{N_l}, \rho, \varepsilon)$ then $\mathit{KPTree}(M, \rho, \sigma)$ is the tree whose root is labeled b, and whose daughters are (in that order) $\mathit{KPTree}(N_1, \rho, \varepsilon)$, ..., $\mathit{KPTree}(N_l, \rho, \varepsilon)$. Otherwise $\mathit{KPTree}(M, \rho, \sigma)$ is ω. Given a closed η-long term M of type 0 in canonical form, we write $\mathit{KPTree}(M)$ for $\mathit{KPTree}(\mathit{unc}(M), \emptyset, \varepsilon)$. We do not enter into the details of the proof of the following lemma which are rather tedious and uninteresting, while its statement is as expected.

Lemma 12. *Given a closed η-long term M of type* 0 *in canonical form, we have that $\mathit{KTree}(M) = \mathit{KPTree}(M)$.*

4.2 Simulation

We let $m + 1$ be the complexity of M, and we construct an m-CPDA $\mathcal{A} = (\Sigma, \Gamma, Q, F, \delta)$ generating $BT(M)$. The main part of the construction of $\mathcal{A}$ is how to represent the environment of the Krivine machine directly in the higher-order stack. The top-most 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 obtained simply by poping this top-element. Non-recursive variables with the highest type order (that is m) 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 as well as the *collapse* operation.

The stack alphabet Γ of $\mathcal{A}$ is the set of non-recursive variables of M together with tuples of terms that are arguments of some subterm of M. While the set Q of states of $\mathcal{A}$ is the set of subterms of M (excluding the arguments of the Y combinator). Before we give the transition rules of $\mathcal{A}$, we explain the way the environment of the Krivine machine is represented as a higher-order stack. We are going to define the function *value* as in 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:

$$value(\pi_i(z), S) = \begin{cases} (N_i, pop_1(S')) & \text{if } top(S) = z,\ S' = collapse(S) \\ & \text{and } top(S') = \overrightarrow{N_l},\ N_i = \pi_i(\overrightarrow{N_l}), \\ value(\pi_i(z), pop_1(S)) & \text{otherwise} \end{cases}$$

Therefore to find the value associated to $\pi_i(z)$, it suffices to search in the topmost first order stack the first occurrence of z with the pop_1 operation, then use the *collapse* operation, and, using the index of the projection, for the new state select the appropriate term in the tuple on top of the stack we have obtained, and finally erase that tuple from the stack using a pop_1 operation to restore the environment of the closure.

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

Notice that, unsurprisingly, the way closures are constructed in $\rho[S]$ is similar to the definition of $value(\pi_i(x), S)$. Notice also that non-recursive variables of maximal order, that is of order m, 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 of $copy_d$ when $d > 1$, so we have $\rho[S] = \rho[copy_d(S)]$.

Definition 9. *Let M_0 be a closed term of type 0. We suppose that M_0 is in the canonical and uncurrified forms. Let m_0 be the highest order of non-recursive variable appearing in M_0. We define a m_0-CPDA $\mathcal{B}(M_0)$ whose states and stack alphabet are subterms of M_0. To describe the transition function of $\mathcal{B}(M_0)$ we describe the actions it performs while having a stack S depending on the state it is in.*

1. *In a state $\pi_i(z)$, there are two possibilities:*
   (a) in case $top(S) = z$, then the automaton goes to state (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 case $top(S) = y$ with $y \neq z$ and then the automaton goes to the new configuration $(\pi_i(z), S')$, where $S' = pop_1(S)$ when $order(y) < m_0$ and $S' = pop_1(pop_1(S))$ when $order(y) = m_0$.
2. *In a state $\lambda x.M$, the automaton goes to the configuration (M, S') where $S' = push^{x,p}(S)$ and $p = m - order(x) + 1$.*

3. *In a state $M\overrightarrow{N_l}$, the automaton goes in the configuration (M, S''') where $S''' = pop_1(S'')$, $S'' = copy_p(S')$ and $S' = push^{\overrightarrow{N_l},1}(S)$ (where $p = m - order(\overrightarrow{N_l}) + 1$).*
4. *In state $\mathbf{x}$, $\mathbf{x}$ being a Y-variable, the automaton goes in the configuration $(term(\mathbf{x}), S)$.*
5. *In state $Y\mathbf{x}.P$, the automaton goes to the configuration (P, S).*
6. *In state b where b is a tree constant of arity l, the automaton goes to $(b, fq_1, \dots fq_l)$. From a state qf_i and stack S it goes to $(N_i, pop_1(S))$ where $\overrightarrow{N_l} = top(S)$. This move implements outputting the constant an going to its arguments.*

Let us briefly explain these rules.

The first case is about looking up the value of the variable: it implements operation *value*. If the value is on the top of the stack we just recover this value, as we have explained when describing encoding of the environment in the stack. If the value 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, 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 pop_1 operations.

In a state $\lambda x.M$ the automaton implements the binding of a closure by 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 $pop_p(S)$.

In a state $M\overrightarrow{N_l}$, 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 $p = m - order(\overrightarrow{N_l}) + 1$ and remove with pop_1 the topmost occurrence of $\overrightarrow{N_l}$ created by the copy.

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

Definition 10. *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. *$\rho = \rho[S]$.*
2. *When σ is nonempty then $\sigma = (\overrightarrow{N_l}, \rho[S''])$ where $S'' = pop_1(S')$, $\overrightarrow{N}_l = top(S')$ and:*

$$S' = \begin{cases} pop_{m-order(P)+2}(S) \text{ when } order(P) < m+1 \\ S \text{ when } order(P) = m+1 \end{cases}$$

3. *For every variable x in S, the collapse pointer at x is of order $m - order(x) + 1$.*

Let's 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 is higher-order, because we only evaluate terms of atomic types, 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 $\mathit{push}^{x^\alpha,p}$ operation with $p = m - \mathit{order}(\alpha)+1$. But $\mathit{order}(\alpha) = \mathit{order}(P)-1$ so that $p = m-\mathit{order}(P)+2$. Thus we shall be able to retrieve the closure, simply by applying a $\mathit{pop}_{m-\mathit{order}(P)+2}$ operation. Nevertheless, this only works when $\mathit{order}(P) < m+1$; indeed if $\mathit{order}(P) = m+1$ then $p = 1$ and the binding is made with a $\mathit{push}^{x^\alpha,1}$ operation, meaning that the closure is on top of the topmost 1-stack of the automaton. This explains why the case where $\mathit{order}(P) = m+1$ is treated differently.

Along this observation we can make a further remark on the case where the state is $\pi_i(z)$ and $\mathit{top}(S) = z$. In this case, the automaton uses a *collapse* operation, nevertheless, as, because of typing again, the variable z has to have been pushed on the stack with a $\mathit{push}^{z,p}$ operation where $p = m - \mathit{order}(z) + 1$, the *collapse* operation is a sequence of pop_p operations. But the closure that has been prepared so as to be bound to the variable λ-abstracted in the term substituted for $\pi_i(z)$ is of an order d strictly smaller than that of z and so, the closure is accessible with a pop_{m-d} operation. But, since $m-d > m-p$, we have the guaranty that the closure is not erased by the *collapse* operation and that it remains accessible with a pop_{m-d} after the *collapse* operation has been performed.

We can now prove that $\mathcal{A}$ simulates the Krivine machine in a similar manner as it was done in the previous translation and obtain the following Lemma:

Lemma 13. *Given a well-typed configuration (P, S) of $\mathcal{A}$ representing (P, ρ, σ), if from (P, S), $\mathcal{A}$ reaches (P_1, S_1) in one step, then (P_1, S_1) is well-typed, it represents (P_1, ρ_1, σ_1) and $(P, \rho, \sigma) \to (P_1, \rho_1, \sigma_1)$.*

Proof. This Lemma is proved by case analysis on the shape of P. It does not present any difficulty, the fact that (P, S) is well-typed implies easily that (P_1, S_1) is also well-typed. Moreover well-typedness guaranties, that the use of *collapse* for retrieving the closure associated to a variable does not destroy in the higher-order stack of the closure representing the stack of the Krivine machine.

Theorem 4. *Consider terms and automata over a tree signature Σ. For every term M of type 0 in the canonical form: if M has the complexity $m+1$ then m-CPDA $\mathcal{B}(\mathit{unc}(M))$ is such that $BT(\mathcal{B}(\mathit{unc}(M))) = \mathit{CTree}(\mathcal{A})$.*

Proof. Using Lemma 8 we consider $\mathit{KTree}(M)$ instead of $BT(M)$. But with Lemma 12 we may use $\mathit{KPTree}(M)$ instead. The tree $\mathit{KPTree}(M)$ starts with the execution of a Krivine machine from $(\mathit{unc}(M), \emptyset, \varepsilon)$. On the other side $\mathit{CTree}(\mathcal{B}(\mathit{unc}(M)))$ starts from configuration consisting of the state $\mathit{unc}(M)$ and stack $\perp$. It is clear that $(\mathit{unc}(M), \perp)$ is well-typed and that it represents $(\mathit{unc}(M), \emptyset, \varepsilon)$. Repeated applications of Lemma 11 give us that either both trees consist of the root labeled with ω, or on *KPTree* side we reach a configuration $(b\overrightarrow{N_l}, \rho, \varepsilon)$ and on the *CTree* side a well-typed configuration $(b\overrightarrow{N_l}, S)$ representing $(b\overrightarrow{N_l}, \rho, \varepsilon)$. Then on the side of *KPTree* the machine produces a tree whose root is b is produced and whose daughters are $KPTree(N_1, \rho, \varepsilon)$, ..., $KPTree(N_l, \rho, \varepsilon)$. On the *CTree* side the automaton produces a tree whose root is b is produced and whose daughters are $CTree(N_1, S)$, ..., $CTree(N_l, S)$. But since $(b\overrightarrow{N_l}, S)$ representing $(b\overrightarrow{N_l}, \rho, \varepsilon)$, we get that for every i in $[1; l]$, (N_i, S) represents (N_i, ρ, ε). Repeating this argument ad infinimum we obtain that the trees $KTree(M)$ and $CTree(\mathcal{B}(unc(M)))$ are identical. □

References

1. K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In *TLCA'05*, volume 3461 of *LNCS*, pages 39–54, 2005.
2. A. Aho. Indexed grammars – an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968.
3. H. Barendregt. The type free lambda calculus. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter D.7, pages 1091–1132. North Holland, 1977.
4. H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1985.
5. H. Barendregt and J. W. Klop. Applications of infinitary lambda calculus. *Inf. Comput.*, 207(5):559–582, 2009.
6. A. Carayol and O. Serre. Collapsible pushdown automata and labeled recursion schemes equivalence, safety and effective selection. In *LICS*, pages 165–174, 2012.
7. A. Carayol and S. Wöhrle. The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *FSTTCS'03*, volume 2914 of *Lecture Notes in Computer Science*, pages 112–124, 2003.
8. D. Caucal. On infinite terms having a decidable monadic theory. In *MFCS'02*, volume 2420 of *Lecture Notes in Computer Science*, pages 165–176, 2002.
9. W. Damm. The IO– and OI–hierarchies. *Theoretical Computer Science*, 20(2):95–208, 1982.
10. M. Dezani-Ciancaglini, E. Giovannetti, and U. de' Liguoro. Intersection Types, Lambda-models and Böhm Trees. In *MSJ-Memoir Vol. 2 "Theories of Types and Proofs"*, volume 2, pages 45–97. Mathematical Society of Japan, 1998.
11. M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS'08*, pages 452–461. IEEE Computer Society, 2008.
12. Y. Ianov. The logical schemes of algorithms. In *Problems of Cybernetics I*, pages 82–140. Pergamon, Oxford, 1969.
13. K. Indermark. Schemes with recursion on higher types. In *MFCS'76*, volume 45 of *LNCS*, pages 352–358, 1976.
14. A. Kfoury and P. Urzyczyn. Finitely typed functional programs, Part II: comparisons to imperative languages. Technical report, Boston University, 1988.
15. T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS' 02*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222, 2002.

16. T. Knapik, D. Niwinski, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars and panic automata. In *ICALP'05*, volume 3580 of *Lecture Notes in Computer Science*, pages 1450–1461, 2005.
17. N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL'09*, pages 416–428. ACM, 2009.
18. J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.
19. A. Maslov. The hierarchy of indexed languages of an arbitrary level. *Soviet. Math. Doklady*, 15:1170–1174, 1974.
20. A. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 12:38–43, 1976.
21. R. Milner. Models of LCF. Memo AIM-186, Stanford University, 1973.
22. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS'06*, pages 81–90, 2006.
23. P. Parys. On the significance of the collapse operation. In *LICS'12*, 2012.
24. G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.