Simple Models for Recursive Schemes

Igor Walukiewicz*

LaBRI, CNRS/Universit Bordeaux, France

Abstract. Higher-order recursive schemes are abstract forms of programs where the meaning of built-in constructs is not specified. The semantics of a scheme is an infinite tree labeled with built-in constructs. The research on recursive schemes spans over more than forty years. Still, central problems like the equality problem, and more recently, the model checking problem for schemes remain very intriguing. Even though recursive schemes were originally though of as a syntactic simplification of a fragment of the lambda calculus, we propose to go back to lambda calculus to study schemes. In particular, for the model checking problem we propose to use standard finitary models for the simply-typed lambda calculus.

1 Introduction

A recursive scheme is a set of equations over a fixed functional signature. On the left of an equation we have a function symbol and on the right a term that is its intended meaning. Consider the following examples borrowed from [10]:

$$\begin{aligned}\mathbf{F}(x) &\equiv \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } \mathbf{F}(x-1)\cdot x,\\ \mathbf{F}(x) &\equiv C\big((Zx),\ A,\ M(\mathbf{F}(P(x)),x)\big).\end{aligned}$$

The first is the usual recursive definition of factorial. The second is the same definition in an abstract form where abstract function names have been used instead of the ones with a well established meaning. Observe that the reverse function

$$Rev(x) \equiv \textbf{if } x = \text{nil } \textbf{then } \text{nil } \textbf{else } \textbf{append}(Rev(\text{tl}(x)), hd(x))$$

has the same abstract form as factorial: the pattern of function calls is the same in the two cases.

The above schemes are of order 1 as functions they define work on elements of a basic type. An order 2 scheme allows to express for example *map* function that applies a given function f to every element of the list l:

$$map(f,x) \ \equiv\ \textbf{if } l = nil \textbf{ then } nil \textbf{ else } \textbf{cons}(f(\textbf{head}(l)), map(f, \textbf{tail}(l)))$$

This is a scheme of order 2 because its argument f is a function on elements of a base type. Higher-order recursive schemes are schemes of arbitrary finite order.

* Supported by ANR 2010 BLAN 0202 01 FREC.

B. Rovan, V. Sassone, and P. Widmayer (Eds.): MFCS 2012, LNCS 7464, pp. 49–60, 2012.

Recursive schemes are about abstract forms of programs where the meaning of constants is not specified. In consequence, the meaning of a scheme is a potentially infinite tree labelled with constants obtained from the unfolding of the recursive definition. Let us immediately note that we impose a typing disciple on terms. In this form recursive schemes are essentially a different presentation of simply typed lambda calculus with fixpoint operators. Indeed, recursive definitions as in the examples above can be reformulated using the fix point operator explicitly.

Recursion schemes were originally proposed by Ianov as a canonical programming calculus for studying program transformation and control structures [18]. The study of recursion on higher types as a control structure for programming languages was started by Milner [30] and Plotkin [34]. Program schemes for higher-order recursion were introduced by Indermark [19]. Higher-order features allow for compact high-level programs. They have been present since the beginning of programing, 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 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].

Research on higher-order schemes has spanned many decades. Recursive schemes appear as an intermediate step in semantics of programming languages. Indeed to compute the semantics of a program one can first compute the infinite tree of behaviors it generates, and then apply an interpretation operation giving the meaning of constants [38,31]. This view brought to the scene the equality problem for schemes [27,11,9,39], whose decidability in the higher-order case is still open. It has then been discovered that schemes have many links with language theory, in particular with context-free and context-sensitive languages [16,9,13]. More recently, it has been understood that recursive schemes give important classes of trees with decidable MSOL theory which makes them interesting from the point of view of automatic verification [21,32].

Let us see an example illustrating why recursive schemes are valuable abstractions of programs. Simple while loops with the usual arithmetic operations are Turing complete. This of course does not make all other programming constructs obsolete. Yet, in the light of this universality result we need to look for some other convincing framework where we could show that concepts like recursive procedures or higher-order types bring something new. The idea is to abstract from the meaning of build-in operations, or in other words to consider them as uninterpreted function symbols. This way a program is stripped to its control structure: it becomes a program scheme. Coming back to our example with while programs, once influence of the arithmetic is stripped away, one can

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

formally show that recursive procedures are indeed more powerful than a simple while loop.

The above example shows that recursive schemes are an insightful intermediate step in giving a denotational semantics of a program. This makes the study of equality or verification problems for schemes interesting. Especially in view that these problems may be decidable for schemes while they are not for interpreted programs.

Equality Problem. Given two schemes decide if they generate the same tree.
Model-Checking Problem. Given a schema and a formula of monadic second-order logic decide if the formula holds in the tree generated by the scheme.

Equality of two schemes gives a provably correct program transformation rule. Equality of first-order schemes is equivalent to equality of deterministic pushdown automata [9]. Hence it is decidable by the result of Sénizergues [39]. Some properties of a program can be expressed in terms of the tree generated by the associated scheme. For example, resource usage patterns can be formulated in fragments of monadic second-order logic and verified over such trees [23]. This is possible thanks to the fact that MSOL model checking is decidable for trees generated by higher-order recursive schemes [32].

The meaning of a recursive scheme is an infinite ranked tree. A natural question is then what are these trees. Are there other characterizations of these objects, and what are their properties? First answers came from language theory. Damm [13] 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 [3,28]. Those languages in turn have been know to be equivalent to higher-order pushdown automata of Maslov [29]. Later it has been shown that trees generated by higher-order safe schemes are the same as those generated by higher-order pushdown automata [21]. This gave rise to so called Caucal 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 [13], and only brought on the front stage by Knapik, Niwiński, and Urzyczyn [21]. Secondly, because it required new insights in the nature of higher-order computation. Pushdown automata have been extended with so called panic operation [22,2]. 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 characterise recursive schemes at all levels [17]. 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 [33].

In this paper we will concentrate on the model checking problem. There has been a steady progress on this problem, to the point that there are now tools implementing verification algorithms for higher-order programs [24]. In contrast, the most classical problem, namely the equivalence problem, remains as a great challenge. The fundamental result of Sénizergues [39] and subsequent revisits of

the proof by Stirling [41], Sénizergues [40], and Jancar [20] give an algorithm to test equivalence of schemes of order 1. Yet this algorithm is not only non-elementary but also gives a strong impression that its average performance would be close to non-elementary too. In contrast, while also non-elementary in the worst case, the algorithms for model checking are capable of solving nontrivial verification problems. Let us also recall that we know that the complexity of the model checking problem is non-elementary [15,6,25], while no non-trivial lower bounds are known for the equivalence problem.

The objective of this short paper is to propose an approach to the model checking problem for schemes. The idea is to work with a richer syntax of simply typed lambda calculus with fixpoints, and to construct suitable finitary models that give the answer to the problem. We will carry out this program only for a relatively small subclass of all monadic-second order properties: the same as considered by Aehlig [1]. The next section introduces necessary notions, in particular that of a Böhm tree of a term. In the following section we show how, given an automaton expressing the property, to construct a desired finitary model from which we can read properties of Böhm trees of terms.

2 Simply Typed Lambda Calculus and Recursive Schemes

Instead of introducing higher-order recursive schemes directly we prefer to start with simply typed lambda calculus with fixpoints, *λY-calculus*. The two formalisms are essentially equivalent for the needs of this paper, but we will prefer work with the later one. It gives us an explicit notion of reduction, and brings the classical notion of Böhm tree [4] 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 $(\alpha \to \beta)$. The order of a type is defined by: $order(0) = 1$, and $order(\alpha \to \beta) = max(1 + order(\alpha), order(\beta))$.

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 are constants ω^α and $Y^{(\alpha\to\alpha)\to\alpha}$. A constant $Y^{(\alpha\to\alpha)\to\alpha}$ will stand for a fixpoint operator, and ω^α for undefined. Of special interest to us will be *tree signatures* where all constants other than Y and ω have order at most 2. Observe that types of order 2 have the form $0^i \to 0$ for some i; the later is a short notation for $0 \to 0 \to \cdots \to 0 \to 0$, where there are $i + 1$ occurrences of 0.

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 $x^\alpha, y^\alpha, \ldots$ that are also terms of type α. If M is a term of type β and x^α a variable of type α then $\lambda x^\alpha.M^\beta$ is a term of type $\alpha \to \beta$. 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 usual operational semantics of the λ-calculus is given by β-reduction. To give the meaning to fixpoint constants we use δ-reduction ($\to_\delta$).

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

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

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 [4,14,5]. We define it below.

A *Böhm tree* is an unranked ordered, and potentially infinite tree with nodes labelled by ω^α or terms of the form $\lambda x_1 \ldots x_n.N$; 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 1. *A* Böhm tree *of a term M is obtained in the following way.*

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

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 except 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 labelled with constants of type $0^k \to 0$.*

Higher-order recursive schemes use 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 operation. 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 $\lambda \boldsymbol{x}.M_F$ where: (i) M_F is an applicative term, (ii) the type of $\lambda \boldsymbol{x}.M_F$ is the same as the type of F, and (iii) the free variables of M are among $\boldsymbol{x}$ and $\mathcal{N}$. For example, the following is a scheme of the map function:

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

The translation from a recursive scheme to a lambda-term is given by a standard variable elimination procedure, using the fixpoint operator 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:

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

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

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

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

We will not recall here a rather lengthy definition of a tree generated by a recursive scheme referring the reader to [21,13]. 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 following.

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.*

3 Models for Model Checking

As we have said in the introduction, in order to study the model checking problem it is very helpful to understand better what are trees generated by recursive schemes. The proof of Ong of decidability of the model checking [32] relies on game semantics to understand the structure of such trees. Later proofs rely on higher-order pushdown automata with panic [22,2], for order 2, and for collapse for all orders [17]. Krivine machines give also a good representation of trees, and another proof of Ong's result [37]. In all these approaches to model checking, one first obtains a characterisation of trees generated by recursive schemes and then solves the model checking problem using this characterisation.

There is another, more denotational, way of approaching the model checking problem. One can analyse the scheme from the point of view of a given property without constructing the generated tree first. This approach can be carried out with a help of a rich typing discipline. For a given property one constructs a set of types and typing rules such that the scheme is typable if and only if the tree it generates satisfies the property. This approach has been successfully carried out for properties expressed by automata with a trivial acceptance condition [23]. The extension to all parity automata required much more work and complicated substantially the approach [26]. It is worth mentioning that the discovery that model checking with respect to automata with trivial conditions is much easier than the general case is due to Aehlig [1]. His proof uses a mixture of semantics and typing systems. The approach used by Kobayashi [23] is based on intersection types refining simple types that allows to capture regular properties of terms. This kind of technique has been probably initiated by Salvati [35].

Instead of typings we propose to use models of simply-typed lambda calculus. For simply-typed lambda calculus without fixpoints the intersection types and standard models are in some sense dual [36]. Our construction hints that this can be also the case in the presence of fixpoint operators. Moreover the model based approach has a striking simplicity in the case we present here.

Since the translation from recursive schemes to the λY-calculus is very direct, it is straightforward to interpret a recursive scheme in a model of the λY-calculus. Once this step is taken, it is even more natural to work directly with the lambda calculus. So starting with a property we would like construct a model such that the value of a term in the model determines if the Böhm tree of the term satisfies the property. This is formalised in Theorem 1 below.

Let us consider finitary models of λY-calculus. We concentrate on those where Y is interpreted as the greatest fixpoint.

Definition 2. *A* GFP-model *of a signature Σ is collection of finite complete lattices, one for each type, together with a valuation of constants: $\mathcal{D} = \langle \{D^\alpha\}_{\alpha \in \mathcal{T}}, \rho \rangle$. The model is required to satisfy the following conditions:*

- *D^0 is a finite lattice;*
- *for every type $\alpha \to \beta \in \mathcal{T}$, $D^{\alpha \to \beta}$ is the lattice of monotone functions from D^α to D^β ordered coordinatewise;*
- *If $c \in \Sigma$ is a constant of type α then $\rho(c)$ is an element of D^α. For every $\alpha \in \mathcal{T}$ it must be that case that $\rho(\omega^\alpha)$ is the greatest element of D^α. Moreover, $\rho(Y^{(\alpha \to \alpha) \to \alpha})$ should be a function assigning to every function $f \in D^{\alpha \to \alpha}$ its greatest fixpoint.*

Observe that every D^α is finite, hence all the greatest fixpoints exists without any additional assumptions on the lattice.

A *variable assignment* is a function υ associating to a variable of type α an element of D^α. If d is an element of D^α and x^α is a variable of type α then $\upsilon[d/x^\alpha]$ denotes the valuation that assigns d to x^α and that is identical to υ otherwise.

The *interpretation of a term M* of type α in the model $\mathcal{D}$ under valuation υ is an element of D^α denoted $[\![M]\!]^\upsilon_{\mathcal{D}}$. The meaning is defined inductively:

- $[\![c]\!]^{\upsilon}_{\mathcal{D}} = \rho(c)$
- $[\![x^\alpha]\!]^{\upsilon}_{\mathcal{D}} = \upsilon(x^\alpha)$
- $[\![MN]\!]^{\upsilon}_{\mathcal{D}} = [\![M]\!]^{\upsilon}_{\mathcal{D}}[\![N]\!]^{\upsilon}_{\mathcal{D}}$
- $[\![\lambda x^\alpha.M]\!]^{\upsilon}_{\mathcal{D}}$ is a function mapping an element $d \in D^\alpha$ to $[\![M]\!]^{\upsilon[d/x^\alpha]}_{\mathcal{D}}$.

As usual, we will omit subscripts or superscripts in the notation of the semantic function if they are clear from the context.

Of course a GFP model is sound with respect to $\beta\delta$-equality. Hence two equal terms have the same semantics in the model. For us it is important that a stronger property holds: if two terms have the same Böhm trees then they have the same semantics in the model.

Lemma 3. *For every GFP-model $\mathcal{D}$, and closed terms M, N of λY-calculus: if $BT(M) = BT(N)$ then $[\![M]\!]_{\mathcal{D}} = [\![N]\!]_{\mathcal{D}}$.*

Before stating the theorem we need to explain how properties of Böhm trees are specified. We will consider tree signatures, so Böhm trees of closed terms of type 0 are just ranked, potentially infinite trees (cf. Lemma 1). To express properties of such trees we can use monadic second-order logic, or an equivalent formalism of finite automata on infinite trees. We will use the later since it allows for an easy formulation of the important restriction we will make. We will consider only automata with trivial acceptance condition. This means that every run of such an automaton is accepting. So the trees that are not accepted are those over which the automaton does not have a run. We define this concept below.

Let us fix a tree signature Σ. Recall that this means that apart from ω and Y all constants have order at most 2. For simplicity of notation we will assume that constants of order 2 have type $0 \to 0 \to 0$. In this case, by Lemma 1, Böhm trees are potentially infinite binary trees. Let Σ_1 be the set of constants of order 1, hence of type 0, and Σ_2 the set of constants of order 2, hence of type $0 \to 0 \to 0$.

Definition 3. *A* finite automaton with trivial acceptance condition *over the signature $\Sigma = \Sigma_1 \cup \Sigma_2$ is*

$$\mathcal{A} = \langle Q, \Sigma, q^0 \in Q, \delta_1 : Q \times \Sigma_1 \to \{ff, tt\}, \delta_2 : Q \times \Sigma_2 \to \mathcal{P}(Q^2)\rangle$$

where Q is a finite set of states and $q^0 \in Q$ is the initial state.

Observe that the type of δ_1 logically follows from the fact that a constant of type 0 is a "function with no arguments", hence the range of δ_1 should be $\mathcal{P}(Q^0)$ that is more intuitively presented as $\{ff, tt\}$.

Automata will run on Σ-labelled binary trees that are partial functions $t : \{0,1\}^* \to \Sigma$ such that their domain is a binary tree, and $t(u) \in \Sigma_1$ if u is a leaf, and $t(u) \in \Sigma_2$ otherwise.

A *run of $\mathcal{A}$ on t* is a partial mapping $r : \{0,1\}^* \to Q$ with the same domain as t an such that:

- $r(\varepsilon) = q^0$, here ε is the root of t.
- $(r(u0), r(u1)) \in \delta_2(t(u), r(u))$ if u is an internal node.

A run is *accepting* if for every leaf u of t either $\delta_1(r(u), t(u)) = tt$, or $t(u) = \omega^0$. The later condition means that the automaton accepts in leaves labelled by ω^0 constant. A tree is *accepted by* $\mathcal{A}$ if there is an accepting run on the tree. The *language* of $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of trees accepted by $\mathcal{A}$.

Observe that our automata have acceptance conditions on leaves, expressed with δ_1, but do not have acceptance conditions on infinite paths. The clause about always accepting in leaves labelled ω^0 has some consequences. In particular, with these automata we cannot define a set of terms that do not have ω^0 in their Böhm tree. This restriction appears also in the works of Aehlig [1] and Kobayashi [23].

Theorem 1. *For every automaton with trivial acceptance conditions $\mathcal{A}$ there is a GFP-model $\mathcal{D}_\mathcal{A}$ and a set $F_\mathcal{A} \subseteq D^0_\mathcal{A}$ such that for every closed term M of type 0:*

$$BT(M) \in L(\mathcal{A}) \quad \textit{iff} \quad [\![M]\!]_{\mathcal{D}_\mathcal{A}} \in F_\mathcal{A}.$$

For $\mathcal{D}_\mathcal{A}$ we take a GFP model with the domain for the base type being the set of subsets of the set of states of $\mathcal{A}$: $D^0_\mathcal{A} = \mathcal{P}(Q)$. This choice determines all D^α. It remains to define interpretation of constants other than ω or Y. A constant $c \in \Sigma$ of type 0 is interpreted as the set $\{q : \delta_1(q, c) = tt\}$. A constant $a \in \Sigma$ of type $0 \to 0 \to 0$ is interpreted as the function whose value on $(S_0, S_1) \in \mathcal{P}(Q)^2$ is $\{q : \delta_2(q, a) \in S_0 \times S_1\}$. Finally, we put $F_\mathcal{A} = \{S : q^0 \in S\}$; recall that q^0 is the initial state of $\mathcal{A}$. The proof of the theorem is rather easy. Lemma 3 allows to work with Böhm trees instead of terms. Then one can use approximations of an infinite tree by its finite prefixes.

Corollary 1. *Given a closed λY-term M of type 0 and an automaton with a trivial acceptance condition $\mathcal{A}$ it is decidable if $BT(M)$ is accepted by $\mathcal{A}$.*

The decidability follows immediately from Theorem 1, and the fact that $\mathcal{D}_\mathcal{A}$ is a finitary model, so the semantics of a term can be computed just using the definition. The answer is positive if an only if the obtained meaning is in $F_\mathcal{A}$. Of course computing the meaning of a term may be computationally difficult. The sizes of domains grow fast with the order of the type: the size of $D^{\alpha\to\beta}_\mathcal{A}$ is exponentially bigger than the size of $D^\alpha_\mathcal{A}$. It is known that the model checking problem is nonelementary [15,6,25], and at closer inspection the worst case complexity of the approach presented here is on a par with other approaches.

It is not clear how to extend this method to automata with parity conditions. What is straightforward to do is to extend it to automata that are dual to automata with trivial acceptance conditions. Automata with trivial acceptance conditions are in fact equivalent in expressive power to alternating parity automata whose all states have rank 0. The dual automata are alternating automata whose all states have rank 1. Let us call them *rank* 1 *automata*. In particular rank 1 automata accept the complements of the languages accepted by automata with trivial conditions.

In the theorem above, we have used GFP models: we have interpreted Y constants as the greatest fixpoint operators. To capture the power of rank 1

automata we take LFP models where Y's are interpreted as the least fixpoint operators, and ω's as the least elements in corresponding lattices. Dualizing the argument we get the corresponding result

Theorem 2. *For every rank* 1 *automaton* $\mathcal{A}$ *there is a LFP model* $\mathcal{E}_{\mathcal{A}}$ *and a set* $F_{\mathcal{A}} \subseteq E^0$ *such that for every closed term* M *of type* 0*:*

$$BT(M) \in L(\mathcal{A}) \quad \textit{iff} \quad [\![M]\!]_{\mathcal{E}_{\mathcal{A}}} \in F_{\mathcal{A}}.$$

4 Conclusions

We have argued that recursive schemes are a fundamental notion that has appeared in a number of areas of computer science like: schematology [27], language theory [13], and verification [21,32]. For this reason there is a number of different approaches to view and study schemes. We have suggested yet another one in this paper. The mixture of the lambda calculus, language theory, and semantics makes this subject a promising playground for all three disciplines.

Let us hint yet another, more logic based, approach to understand schemes. Instead of trying to find a new device capable of generating the same trees as schemes do, one can look for operations on trees or graphs. In the case of safe schemes this program has been successfully carried out resulting in what is now called Caucal hierarchy [8]. This is the set of trees obtained from the one node tree by operations of unfolding of a graph into a tree [12], and MSOL interpretations. Since both these operations preserve MSOL decidability, we immediately obtain that all the trees in the Caucal hierarchy have decidable MSOL theory. It can be then shown that these are up to simple MSOL interpretations precisely the trees generated by recursive schemes. A number of other characterisations of this class of trees exist: via rewriting rules, using Muchnik's unfolding operation, using higher order pushdown automata (without collapse) [7]. By the result of Parys [33] we know that there are trees generated by recursive schemes that do not belong to Caucal hierarchy. One can imagine that there exists some operation preserving decidability of MSOL theories that allows to obtain all tress generated by higher-order recursive schemes in the same way as the unfolding operation does for safe schemes.

References

1. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. Logical Methods in Computer Science 3(1), 1–23 (2007)
2. Aehlig, K., de Miranda, J.G., Ong, C.-H.L.: The Monadic Second Order Theory of Trees Given by Arbitrary Level-Two Recursion Schemes Is Decidable. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 39–54. Springer, Heidelberg (2005)
3. Aho, A.V.: Indexed grammars – an extension of context-free grammars. J. ACM 15(4), 647–671 (1968)
4. Barendregt, H.: The type free lambda calculus. In: Handbook of Mathematical Logic, ch. D.7, pp. 1091–1132. North-Holland (1977)

5. Barendregt, H., Klop, J.W.: Applications of infinitary lambda calculus. Inf. Comput. 207(5), 559–582 (2009)
6. Cachat, T., Walukiewicz, I.: The Complexity of Games on Higher Order Pushdown Automata. Internal report
7. Carayol, A., Wöhrle, S.: The Caucal Hierarchy of Infinite Graphs in Terms of Logic and Higher-Order Pushdown Automata. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 112–123. Springer, Heidelberg (2003)
8. Caucal, D.: On Infinite Terms Having a Decidable Monadic Theory. In: Diks, K., Rytter, W. (eds.) MFCS 2002. LNCS, vol. 2420, pp. 165–176. Springer, Heidelberg (2002)
9. Courcelle, B.: A representation of trees by languages I. Theor. Comput. Sci. 6, 255–279 (1978)
10. Courcelle, B.: Recursive applicative program schemes. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 459–492. Elesvier (1990)
11. Courcelle, B., Nivat, M.: Algebraic families of interpretations. In: FOCS (1976)
12. Courcelle, B., Walukiewicz, I.: Monadic second-order logic, graphs and unfoldings of transition systems. Annals of Pure and Applied Logic 92, 35–62 (1998)
13. Damm, W.: The IO– and OI–hierarchies. Theoretical Computer Science 20(2), 95–208 (1982)
14. Dezani-Ciancaglini, M., Giovannetti, E., de' Liguoro, U.: Intersection Types, Lambda-models and Böhm Trees. In: MSJ-Memoir "Theories of Types and Proofs", vol. 2, pp. 45–97. Mathematical Society of Japan (1998)
15. Engelfriet, J.: Iterated push-down automata and complexity classes. In: 15th STOC 1983, pp. 365–373 (1983)
16. Engelfriet, J., Schmidt, E.: IO and OI. Journal of Computer and System Sciences 15(3), 328–353 (1977)
17. Hague, M., Murawski, A.S., Ong, C.-H.L., Serre, O.: Collapsible pushdown automata and recursion schemes. In: LICS 2008, pp. 452–461. IEEE Computer Society (2008)
18. Ianov, Y.: The logical schemes of algorithms. In: Problems of Cybernetics I, pp. 82–140. Pergamon, Oxford (1969)
19. Indermark, K.: Schemes with Recursion on Higher Types. In: Mazurkiewicz, A. (ed.) MFCS 1976. LNCS, vol. 45, pp. 352–358. Springer, Heidelberg (1976)
20. Jancar, P.: Decidability of DPDA language equivalence via first-order grammars. In: LICS 2012 (2012)
21. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-Order Pushdown Trees Are Easy. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)
22. Knapik, T., Niwiński, D., Urzyczyn, P., Walukiewicz, I.: Unsafe Grammars and Panic Automata. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1450–1461. Springer, Heidelberg (2005)
23. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: POPL 2009, pp. 416–428. ACM (2009)
24. Kobayashi, N.: Higher-order model checking: From theory to practice. In: LICS 2011, pp. 219–224 (2011)
25. Kobayashi, N., Ong, C.-H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. Logical Methods in Computer Science 7(4) (2011)

26. Kobayashi, N., Ong, L.: A type system equivalent to modal mu-calculus model checking of recursion schemes. In: LICS 2009, pp. 179–188 (2009)
27. Manna, Z.: Mathematical Theory of Computation. McGraw-Hill (1974)
28. Maslov, A.: The hierarchy of indexed languages of an arbitrary level. Soviet. Math. Doklady 15, 1170–1174 (1974)
29. Maslov, A.: Multilevel stack automata. Problems of Information Transmission 12, 38–43 (1976)
30. Milner, R.: Models of LCF. Memo AIM-186. Stanford University (1973)
31. Nivat, M.: On interpretation of recursive program schemes. In: Symposia Mathematica, vol. 15 (1975)
32. Ong, C.-H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006, pp. 81–90 (2006)
33. Parys, P.: On the significance of the collapse operation. In: LICS 2012 (2012)
34. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. 5(3), 223–255 (1977)
35. Salvati, S.: Recognizability in the Simply Typed Lambda-Calculus. In: Ono, H., Kanazawa, M., de Queiroz, R. (eds.) WoLLIC 2009. LNCS, vol. 5514, pp. 48–60. Springer, Heidelberg (2009)
36. Salvati, S., Manzonetto, G., Gehrke, M., Barendregt, H.: Loader and Urzyczyn Are Logically Related. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part II. LNCS, vol. 7392, pp. 364–376. Springer, Heidelberg (2012)
37. Salvati, S., Walukiewicz, I.: Krivine Machines and Higher-Order Schemes. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 162–173. Springer, Heidelberg (2011)
38. Scott, D.: Continuous lattices. In: Proc. of Dalhousie Conference. Lecture Notes in Mathematics, vol. 188, pp. 311–366. Springer (1972)
39. Sénizergues, G.: L(A)=L(B)? Decidability results from complete formal systems. Theor. Comput. Sci. 251(1-2), 1–166 (2001)
40. Sénizergues, G.: L(A)=L(B)? A simplified decidability proof. Theor. Comput. Sci. 281(1-2), 555–608 (2002)
41. Stirling, C.: Deciding DPDA Equivalence Is Primitive Recursive. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 821–832. Springer, Heidelberg (2002)