Automata theory and higher-order model-checking

Igor Walukiewicz, CNRS, LaBRI, Bordeaux University

In verification, an automata theoretic approach is by now a standard. In order to extend this approach to higher-order programs we need a good understanding of higher-order control flow, and for this semantics has the right tools. We present some results and methods of this subject between automata theory and semantics.

1. INTRODUCTION

As a relatively young science, Computer Science progresses in many different directions. It is particularly pleasant then when two such directions meet. Higher-order verification is becoming a meeting ground for semantics, and automata theory communities. This note intends to present some of the reasons for this common interest.

Rabin's theorem states that monadic-second order theory of the full binary tree is decidable. What about other trees? A *behavior* of a program is a sequence of actions it performs, so the set of behaviors can be arranged into a potentially infinite tree. Such a tree faithfully represents the semantics of the program: it is an infinite term that when evaluated gives the denotational value of the program. What will be of particular interest for us here is that this tree makes it also possible to talk about *behavioral properties* of the program. For example, we may be interested to know if every behavior can be prolonged to reach a ~~finite~~ state. Such a property can be expressed in monadic second-order logic over trees. final

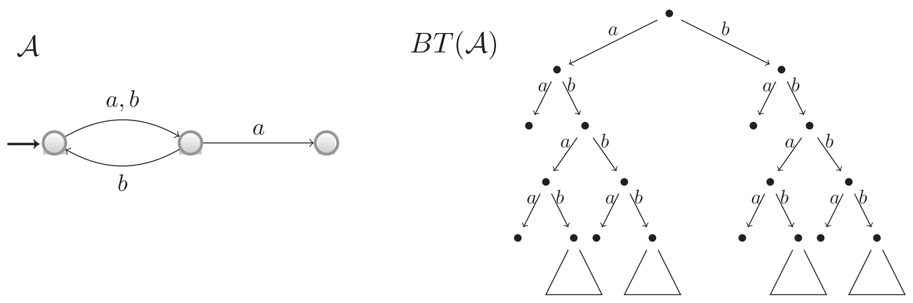
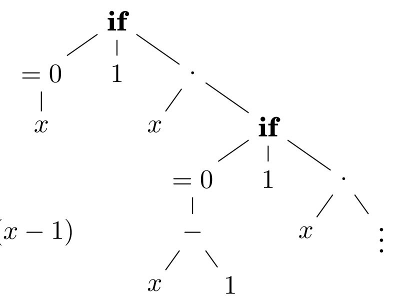


Fig. 1. A finite automaton and its behavior tree.

To start with a simple example of this setting consider deterministic finite automata as programs. Behaviors of an automaton are prefixes of words over which it has a run. These can be arranged in a behavior tree as in the example in Figure 1. One can then consider properties of this tree that are properties of the behaviors of the automaton. For example: is it the case that every path can be prolonged to a path ending in a leaf? In passing let us remark that for a human it is easier to see that this property holds by looking at the automaton rather than at the behavior tree since it suffices to notice that from every state there is a path to a state without outgoing edges.

Trees like $BT(\mathcal{A})$ have appeared very early in the history of computer science. Ianov in late 50-ties [Ianov 1960] has considered recursive schemes whose semantics were such behavior trees; this is about the same time when Rabin and Scott have introduced finite automata. In the beginning of 70-ties, Elgot and Scott [Elgot 1971; Scott 1971] proposed to use free interpretation of programs as an intermediate step in program semantics. In the free interpretation, the control structure of a program is executed but atomic operations are left non-interpreted. The result for a program P is a potentially infinite tree $BT(P)$. This tree, considered as an infinite term, can be then evaluated in a particular domain of interest. An example of the free interpretation is given in Figure 2. Observe that even for this very simple example the behavior tree is not regular: it has infinitely many different subtrees since the length of the branches going to the left increases as -1 operations accumulate.

$BT(Fct(x))$



$Fct(x) \equiv \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \cdot Fct(x-1)$

Fig. 2. Factorial program and its behavior tree.

In λ-calculus, behavior trees are known as *Böhm trees*. These are a sort of normal forms of terms. The difference is that normal forms exist only for terms representing terminating programs while every term has a Böhm tree. In this note we are actually mostly interested in programs that do have nonterminating computations since we see them as representations of reactive programs that interact with their environment ad infinitum. As we will see, the Böhm tree of a closed term of the base type is simply a ranked tree, so we can talk about monadic second-order properties of such trees. In the context of automata theory, Courcelle proposed the metaphor of machines and their behaviors [Courcelle 1995] that we have employed in our example from Figure 1.

We will consider programs written in the λY-calculus, the simply typed λ-calculus with the fixpoint combinator. This is an abstraction of a strongly typed programming language with recursive definitions and higher-order procedures/functions. Higher-order permits to implement continuations in the λY-calculus, and these in turn allow to express a vast variety of control-flow operators. According to our methodology, the constants in the λY-calculus are not interpreted. Yet, using control operators it is possible to simulate finite domains and conditionals over them. Let us also mention that PCF [Plotkin 1977], probably the most studied language in the field, is the λY-calculus extended with integers and basic operations over integers. Under a different syntax the λY-calculus has also been intensively studied by language theoretic community. One can cite the work of Engelfriet and Schmidt on IO and OI [Engelfriet and Schmidt 1977; Engelfriet and Schmidt 1978], or the work of Damm on (safe) recursive schemes [Damm 1982]. At present, the λY-calculus is the most expressive formalism for which the results we report below hold.

We can now come to questions that will interest us in this overview. Given a term M of the λY-calculus we will be interested in properties of the Böhm tree $BT(M)$. These properties will be expressed by a monadic second-order formula φ. We require that M is closed and of the ground type so that $BT(M)$ is just a ranked tree.

The basic question is that of decidability

Decidability. Given a closed term M of the ground type and a formula φ, decide if φ holds in $BT(M)$.

Its particular instance is the question we have started with: the term M can encode a finite automaton, and φ can express the property that every path can be prolonged to a path reaching a leaf.

A more refined version of the decidability question is the transfer property. Its formulation uses the fact that a term M itself can be seen as a graph, so MSOL can be used to define sets of terms.

Transfer. Given φ, construct an MSOL formula $\widehat{\varphi}$ such that for every closed term M of the ground type: $\widehat{\varphi}$ holds in M iff φ holds in $BT(M)$.

The transfer property implies decidability since the graph representation of M is finite, so it is decidable if $\widehat{\varphi}$ holds in M. The formulation of the transfer property comes from a long tradition of studying MSO-compatible operations that we will outline in Section 2 [Blumensath et al. 2008]. The main strength of this formulation comes from the fact that the formula $\widehat{\varphi}$ is constructed uniquely from φ, and is supposed to work for all terms M. We will actually see later that the transfer property holds only when we fix a finite set of variables that can occur in a term.

Another refinement of the decidability question makes an explicit connection between semantics and automata theory.

Recognizability. Given φ, construct a finitary model $\mathcal{D}$ of the λY-calculus and a set $F \subseteq \mathcal{D}$ such that for every closed term M of the base type: φ holds in $BT(M)$ iff $[\![M]\!]^{\mathcal{D}} \in F$.

Here $[\![M]\!]^{\mathcal{D}}$ stands for the value of term M in model $\mathcal{D}$; we will define models formally in Section 3. A model is finitary if for every type it has finitely many functions of this type. We cannot simply require that the model is finite since functions of different types, like $o \to o$ or $o \to (o \to o)$, should be interpreted by different elements. The recognizability property implies decidability: since $\mathcal{D}$ is finitary, one can calculate $[\![M]\!]^{\mathcal{D}}$ and check if it belongs to F. We will see that recognizability implies the transfer property. The most striking fact about the recognizability property is that while being a purely semantic question, it is a generalization of recognizability of languages by finite semi-groups; a standard notion in language theory.

Finally, we would like to mention another question since it has been there from the beginning of the subject.

Equality. Given two closed λY-terms of the base type, M and N, decide if $BT(M) = BT(N)$.

Unfortunately, we do not have a lot to say about this question. It is of a somehow different nature since it does not mention a formula, but rather asks if two terms evaluate to the same result. Program equivalence has been, almost by definition, a central subject in semantics. Decidability of the equality problem for the λY-calculus is still open.

Broader context

The study of recursion on higher types as a control structure for programming languages was started by Milner [Milner 1973] and Plotkin [Plotkin 1977]. Program schemes for higher-order recursion were introduced by Indermark [Indermark 1976]. The λY-calculus we consider here is a different presentation of higher-order recursive schemes.

Research on higher-order schemes has spanned many decades. Recursive schemes appear as an intermediate step in semantics of programming languages [Scott 1972; Nivat 1975]. It has been discovered that schemes have many links with language theory, in particular with context-free and context-sensitive languages [Engelfriet and Schmidt 1977; Courcelle 1978; Damm 1982]. 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 [Knapik et al. 2002; Ong 2006].

The equality of two schemes implies that the two programs they represent have the same meanings in every model. Equality problem for first-order schemes is equivalent to the equality problem for deterministic pushdown automata [Courcelle 1978]. Thus the fundamental result of Sénizergues [Sénizergues 2001] and subsequent revisits of the proof by Stirling [Stirling 2002], Sénizergues [Sénizergues 2002], and Jancar [Jancar 2012] give an algorithm to test equivalence of schemes of order 1.

2. TRANSFER THEOREMS

Rice theorem tells us that no non-trivial property of the behaviour of a Turing machine can be decided by examining the machine itself. But what about simpler machines? For example, reaching an accepting state is a property of the behavior, and it is decidable for numerous computation models. In this section we will be interested in deciding all monadic second-order properties of behaviors. Transfer theorems are powerful tools to obtain such decidability results because they compose. We will sketch a sequence of results culminating in the transfer property for the λY-calculus.

At this point it would be useful to say what is a behavior of a machine. We suppose that an execution of a machine produces a sequence of, non-interpreted, atomic actions. The behavior then is a tree constructed from all such sequences. More abstractly, one can think of a machine as a graph (graph of a finite automaton for example), and behavior as an operation on this graph. *Unfolding* from a given initial node, is an important example of such an operation on graphs. In Figure 3 we see a two node graph and its unfolding that is the full binary tree.

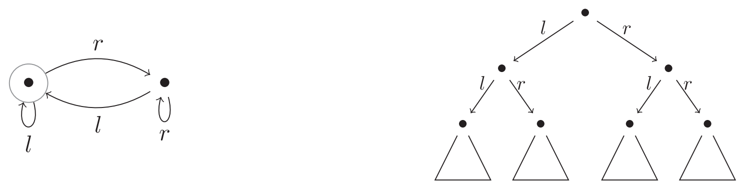


Fig. 3. A two node graph and its unfolding from the initial node marked with a circle.

Monadic second-order logic (MSOL) is an extension of first-order logic with quantification over sets; or said differently, over monadic predicates. We use $x, y, \dots$ to range over individual elements and $X, Y, \dots$ to range over sets. We will interpret MSOL in trees or graphs. The *full* Σ*-tree* has as Σ^* as the set of nodes, with the empty word ε being the root of the tree. The successor relations give the successor of a node for every label: $S_a(w, wa)$ holds for all $a \in \Sigma$ and $w \in \Sigma^*$. A Σ-tree for $\Sigma = \{l, r\}$ is the full binary tree. For example, the formula:

$$\psi(x, x') \equiv \forall Z.\ [Z(x) \wedge (\forall y, y'.\ Z(y) \wedge S_l(y, y') \Rightarrow Z(y'))] \Rightarrow Z(x')$$

says that x' is reachable from x by a sequence of steps to the left, i.e., steps given by S_l relation.

Rabin's theorem says that MSOL theory of the full binary tree is decidable. In other words, there is a procedure that given a sentence of MSOL determines whether this formula holds in the full binary tree. It is not important that the tree is binary, the same result holds for a tree of any fixed branching. If branching is not fixed, or if it is unbounded, then some care is needed to adapt the signature of the logic. If the branching is unary then the tree is actually an infinite sequence. The decidability of MSOL theory of a sequence was proved by Büchi [Büchi 1960]. MSOL is not decidable over the class of all graphs. Yet the MSOL theory of a fixed finite graph is decidable.

The *transfer property* for a graph operation $\mathcal{F}$ says that for every monadic second-order formula φ one can construct a monadic second-order formula $\widehat{\varphi}$ such that for every graph G:

$$G \vDash \widehat{\varphi} \qquad \text{iff} \qquad \mathcal{F}(G) \vDash \varphi$$

Figure 4 gives some intuitions behind this formulation. Machine is represented as a graph G and behavior as a, very irregular, tree $\mathcal{F}(G)$. For example, if $\mathcal{F}$ is the unfolding operation from the initial state and φ is a formula saying that there is a red state, then $\widehat{\varphi}$ is the formula saying that there is a path from the initial state to a red state.

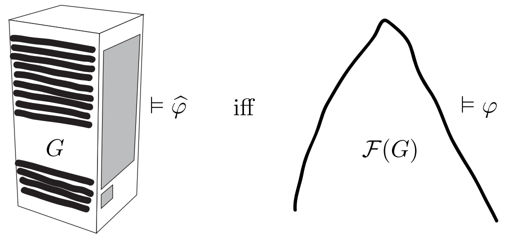


Fig. 4. Transfer property: for a given φ describing a property of behaviors, there is a property $\widehat{\varphi}$ of machines with such behaviors.

Logical interpretation provides the first class of important examples of operations having the transfer property. From an edge colored graph $G = \langle V, \{R_a\}_{a\in\Sigma}\rangle$ we can define another graph $G' = \langle V', \{R'_b\}_{b\in\Sigma'}\rangle$ by means of MSOL formulas. We take one formula $\theta_{dom}(x)$ for the domain, and a formula $\theta_b(x, y)$ for every relation R'_b, with $b \in \Sigma'$. This determines:

$$V' = \{v \in V : G \vDash \theta_{dom}(v)\} \qquad \text{and} \qquad R'_b = \{(v_1, v_2) \in V^2 : G \vDash \theta_b(v_1, v_2)\}$$

Now, given an MSOL formula φ we can construct $\widehat{\varphi}$ by relativizing quantifiers to V' and substituting θ'_b for R'_b:

$$\widehat{\exists x.\psi(x)} \mapsto \exists x.\theta_{dom}(x) \wedge \widehat{\psi} \qquad \widehat{R'_b(x, y)} \mapsto \theta_b(x, y)$$

We get that $G \vDash \widehat{\varphi}$ iff $G' \vDash \varphi$.

MSOL transduction [Courcelle 1994] is a bit more general operation allowing first to make a fixed number of copies of G, and then to use a logical interpretation. Copying of a structure can be simulated in the formula $\widehat{\varphi}$ by the use of different variables. For example a subformula $\forall x.\psi$ in φ is replaced by $\forall x_1, x_2.\widehat{\psi}$ in $\widehat{\varphi}$; with x_1 simulating x over the first copy and x_2 over the second copy. This idea leads to the proof that MSOL transduction has also the transfer property. Actually, one can go even further and consider MSOL transductions with parameters, this leads to a notion nondeterministic transductions as apposed to the deterministic ones that we adopt here.

Finally, the unfolding operation has the transfer property [Semenov 1984; Courcelle and Walukiewicz 1998; Walukiewicz 2002]. This statement implies Rabin's theorem: to decide if a MSOL formula φ holds in the full binary tree, we take $\widehat{\varphi}$ given by the transfer property and check if it holds in the two node graph on the left of Figure 3.

2.1. Transfer for pushdown automata

In order to get some feeling for how transductions and unfoldings can be put to work, we will look at the case of pushdown automata.

A pushdown automaton is given by a set of (prefix rewriting) rules of the form:

$$\text{push: } q\gamma \xrightarrow{a} q'\gamma'\gamma \qquad \text{test: } q\gamma \xrightarrow{a} q'\gamma \qquad \text{pop: } q\gamma \xrightarrow{a} q'$$

where, $q, q' \in Q$ come from a finite set of states, $a \in \Sigma$ from the input alphabet, and $\gamma, \gamma' \in \Gamma$ from the stack alphabet.

The *graph of configurations* of a pushdown automaton has $Q \times \Gamma^*$ as nodes and edges $q\gamma w \xrightarrow{a} q'vw$, for $q\gamma \xrightarrow{a} q'v$ a rule of the automaton and $w \in \Gamma^*$ arbitrary. For example, consider the three rules:

$$q\gamma \xrightarrow{a} q\gamma\gamma \qquad\qquad q\gamma \xrightarrow{b} q'\gamma \qquad\qquad q'\gamma \xrightarrow{c} q'$$

The fragment of the graph of configurations reachable from the initial node $q\gamma$ is presented in Figure 5(a). This graph can be defined inside of $(\Sigma \cup \Gamma)$-tree as shown in Figure 5(b). Thin edges are the edges of the tree, they are labeled with letters from $\Sigma \cup \Gamma$. Thicker edges are the edges of the pushdown graph. For example, edges labeled b link a node that is a target of q edge to its sibling that is a target of q' edge; clearly this can be expressed by an MSOL formula. Generalizing this example one can see that the configuration graph of a pushdown automaton can be defined by an MSOL interpretation inside a $(\Sigma \cup \Gamma)$-tree.

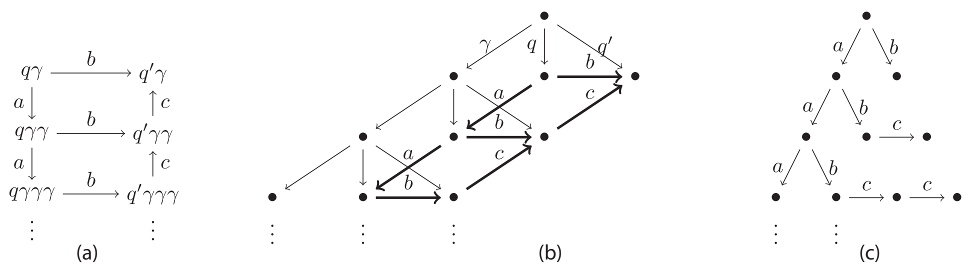


Fig. 5. Pushdown graph, its definition inside $(\Sigma \cup \Gamma)$-tree, and its unfolding into a behavior tree.

The behavior tree of the automaton, Figure 5(c), is obtained by unfolding the configuration graph from the node $q\gamma$. Observe that this is not a regular tree because the length of branches going to the right increases. Since this tree has been obtained from a full tree by an MSOL interpretation and unfolding, it has decidable MSOL theory. This gives the following theorem that has been first proved by Muller and Schupp [Muller and Schupp 1985].

THEOREM 2.1. *For every pushdown automaton, the MSOL theory of its behavior tree is decidable.*

2.2. The λY-calculus

In this section we present the λY-calculus as another way to define infinite trees. In the previous subsection we have considered trees obtained by the unfolding of the graph of configurations of a pushdown automaton. Terms can be seen as another type of machines. The computation rules are β-reduction for application of function arguments, and δ-reduction for fixpoint unfolding. Under these rules a term of this calculus evaluates to a potentially infinite tree, called Böhm tree. It can be seen as the behavior tree of the term considered as a machine. In order to stick to the standard vocabulary, we will refer to behavior trees of terms as Böhm trees.

The class of Böhm trees of λY-terms is quite rich; one of the richest we know of having decidable MSOL theory. It contains behavior trees of pushdown automata considered in the the previous subsection.

Types are essential to what follows since terms in the λY-calculus are typed. The *set of types* $\mathcal{T}$ is constructed from a unique *basic type* o using a binary operation $\to$ that associates to the right. Thus o is a type; and if A, B are types, so is $(A \to B)$. The order of a type is defined by: $order(o) = 0$, and $order(A \to B) = max(1 + order(A), order(B))$. Types of order 1 are of the form $o \to \cdots \to o \to o$ that we abbreviate $o^i \to o$ when they contain $i+1$ occurrences of o. We will consider only *tree signatures* that are finite sets of *typed constants of order at most* 1. This is an important restriction; as we will see later, it implies that Böhm trees of closed terms of the base type are just ranked trees and do not have any λ-binders.

Terms of the λY-calculus are built from the constants in the signature, typed variables $x^A, y^A, \ldots$, and constants Y^A, Ω^A for every type A. The last two stand for the *fixpoint combinator* and *undefined term*, respectively. Bigger terms are constructed using typed application, $M^{A\to B}N^A$, and λ-abstraction, $\lambda x^A.\ M$. For readability we will often omit type annotations. We will also often write $Yx.M$ instead of $Y(\lambda x.M)$.

For example, consider constants $a, b : o \to o$, and $c : o$. Then $\lambda x^o.a(bx)$ is a term. We will denote it M_{ab} as we can think of it as representing a word ab. A more complicated example is $\lambda z_1^{o\to o}\lambda z_2^{o\to o}\lambda x.\ z_1(z_2(x))$ which represents function composition, let us call this term M_{comp}. Its type is $(o \to o) \to (o \to o) \to o \to o$.

The operational semantics of the λY-calculus, is given by *β-contraction*, and *δ-contraction* rules

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

We have for example that $M_{comp}M_{ab}M_{ab}$ reduces to M_{abab}; function composition works as concatenation on terms representing words. A subterm of the form as at the left-hand-side of these two rules is called *β-redex*, or *δ-redex*, respectively. Let us underline that the substitution $M[N/x]$ in the β-reduction stands for capture avoiding substitution: the variables free in N should not be captured by binders in ~~λ~~. Consider for example $(\lambda x.\lambda y.x)y \to_\beta (\lambda y.x)[y/x]$ that is just a projection on the first argument applied to y. Textual substitution would give $\lambda y.y$ which is wrong. The capture avoiding substitution requires to change the name of the bound variable resulting in $\lambda z.y$. M

The reflexive transitive closure $\to_{\beta\delta}^*$ of $\to_{\beta\delta}$ is called *$\beta\delta$-reduction*. This relation is confluent and enjoys subject reduction (*i.e.* the type of a term does not change during reduction). Confluent means that every two sequences of reductions can be extended so that they end in the same term. A term is in a *head normal form* if it is of the form $\lambda\vec{x}.\ N_0N_1\ldots N_k$ with N_0 a variable or a constant, and $\lambda\vec{x}$ a possibly empty sequence of λ-abstractions. Notice that no further reduction can modify $\lambda\vec{x}.\ N_0$ prefix of the head normal form. The confluence property implies that if a term reduces to a term in a head normal form as above then $\lambda\vec{x}$ and N_0 do not depend on what reduction sequence has been used.

To obtain the *Böhm tree* of a term M, denoted $BT(M)$, we find its head normal form $M \to^*_{\beta\delta} \lambda\vec{x}.\ N_0 N_1 \dots N_k$. Then $BT(M)$ is a tree having its root labeled by $\lambda\vec{x}.N_0$ and having $BT(N_1)$, ..., $BT(N_k)$ as subtrees. If M does not have a head normal form then we put $BT(M) = \Omega^o$. Observe that if M is a closed term of type o then since we work with tree signatures, the head normal form must be necessary of the shape $bN_1, \dots, N_k$ with b a constant, and $N_1, \dots, N_k$ terms of the base type. The confluence property of $\to_{\beta\delta}$ reduction guarantees that the Böhm of a term is unique.

The definition implies that a Böhm tree of a closed term of type o over a tree signature is a potentially infinite ranked tree: a node labeled by a constant b of type $o^k \to o$ has k successors. Constants Ω can appear only in leaves, they denote unproductive parts of the computation. A Böhm tree is a kind of a potentially infinite normal form of a λY-term.

As a simple example, let us take $M = \lambda x.ax$. We have a reduction sequence

$$YM \to_\delta (\lambda x.ax)(YM) \to_\beta a(YM) \to^*_{\beta\delta} a(a(YM)) \to \dots$$

So $BT(YM)$ is the infinite sequence $aa\dots$

For a more complicated example take $(Yz.\ N)a$ where $N = \lambda g.g(b(z(\lambda x.g(g\,x))))$. Both a and b have the type $o \to o$; while z has type $(o \to o) \to o$, and so does N. Observe that we are using a more convenient notation Yz here. The Böhm tree of $(Yz.N)a$ is

$$BT((YF.N)a) = aba^2ba^4b\dots a^{2^n}b\dots$$

after every consecutive occurrence of b the number of occurrences of a doubles because of the double application of g inside N.

2.3. Transfer theorem in a simple case

Order 1

To get some feeling for the λY-calculus and the methods we will use, we will consider a relatively simple case of terms whose all the variables are of type o. The method we will use was proposed by Courcelle and Knapik [Courcelle and Knapik 2002]. We will not consider the Y combinator in this subsection as it does not pose particular difficulties. Anyway, all that we will say here applies to infinite terms, so Y can be replaced by an infinite iterator, that is an infinite regular term.

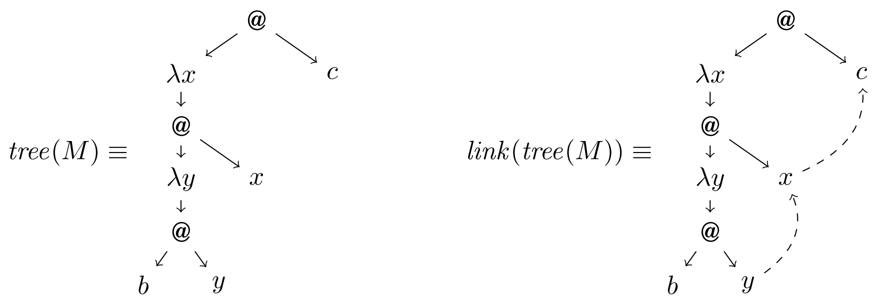


Fig. 6. A term $M = (\lambda x.\ (\lambda y.by)x)c$ represented as a tree, and the same tree with links from variables to arguments; symbol @ stands for application.

We now explain on an example how we can find the Böhm tree of a term M. Terms can be represented as trees in an expected way, cf. Figure 6. We will construct the Böhm tree of M from $tree(M)$ by means of MSOL definable transductions and unfolding. The main technical property of the case we consider in this subsection is that when reducing a β-redex we do not create a new β-redex; this is because we subsitute only terms of the base type. This property implies that after reducing all the redexes in parallel we obtain the Böhm tree.

The first step is to add edges from bound variables to arguments they are applied to. This can be done by finding the λ-binder of a variable, then finding the corresponding application, and linking the variable to the right argument of this application, cf. Figure 6. Let us call this operation $\mathit{link}(\cdot)$.

In the second step one can read out the Böhm tree of M from the unfolding of $\mathit{link}(\mathit{tree}(M))$. It suffices to start in the root and to follow the leftmost path in the $\mathit{link}(\mathit{tree}(M))$ till a constant or an application node whose first argument is a constant; at this point one should take this application node, the constant below, and continue from all other successors of the application node. In our example, the application we find is the one that is the father of b. Then we take b node, and find c by following dashed links from y. Let us call this operation $\mathit{smpl}(\cdot)$. We have $BT(M) = \mathit{smpl}(\mathit{link}(\mathit{tree}(M)))$.

The above description gave some intuitions why $\mathit{smpl}(\mathit{link}(\mathit{tree}(M)))$ can be constructed from $\mathit{link}(\mathit{tree}(M))$ by an unfolding followed by an MSOL transduction. More careful examination would show that $\mathit{link}(\cdot)$ is an MSOL transduction if (i) we fix a finite set of variables, so that we can find the corresponding binder; and (ii) fix the maximal arity of types in order to find the application corresponding to a given binder. To formalize this we introduce a set $\mathit{Terms}(\Sigma, \mathcal{T}, \mathcal{X})$ of terms over signature Σ, using λ binder only on variables from a finite set $\mathcal{X}$, and with all subterms having a type in a finite set $\mathcal{T}$. Actually, if we use $Yx.M$ notation then in this definition we need not limit the use of variables bound by Y. The link operation restricted to $\mathit{Terms}(\Sigma, \mathcal{T}, \mathcal{X})$, is an MSOL transduction. To summarize, $\mathit{smpl}(\mathit{link}(\mathit{tree}(M)))$ is obtained from $\mathit{tree}(M)$ by MSOL transductions and unfolding. Since $\mathit{smpl}(\mathit{link}(\mathit{tree}(M)))$ is $BT(M)$, and since these operations have the transfer property we obtain:

THEOREM 2.2. *Fix Σ, $\mathcal{T}$, and $\mathcal{X}$. For every MSOL formula φ one can effectively construct an MSOL formula $\widehat{\varphi}$ such that for every closed λY-term $M \in \mathit{Terms}(\Sigma, \mathcal{T}, \mathcal{X})$ of type o with all variables of type o:*

$$M \vDash \widehat{\varphi} \qquad \textit{iff} \qquad BT(M) \vDash \varphi.$$

2.4. Transfer for the safe λY-calculus

When computing with $\beta\delta$-reduction it may be necessary to rename bound variables, cf. example on page 17. This corresponds to the concept of static binding in programming languages. Here we will consider *safe* λY-terms whose form guarantees that renaming is not needed during a reduction. For this class one can adopt the method from the previous subsection.

The notion of safety, implicit in the work of Damm [Damm 1982], was explicitly defined by Knapik et al. [Knapik et al. 2002] in the context of recursive schemes. Blum and Ong [Blum and Ong 2009; Blum 2009] have extended this notion to the whole simply typed lambda-calculus and proved many properties about it. After some proper handling of Y [Salvati and Walukiewicz 2016], this notion for the λY-calculus corresponds exactly to the one defined by Knapik et al. for recursive schemes. Below we briefly present the notion of safety, and give an idea why renaming is not necessary when reducing safe terms. Finally, we will sketch how to apply the technique from the previous section to get the transfer property for safe terms.

In order to simplify the definitions we will consider only *homogeneous types*. Every type has the form $A_1 \to \cdots \to A_n \to o$. A type is homogeneous if the orders of $A_1, \ldots, A_n$ form a non-increasing sequence: $\mathit{order}(A_1) \geq \cdots \geq \mathit{order}(A_n)$. Observe that every function can be converted to a function of a homogeneous type by simply reordering its arguments.

We define *safe terms* of a homogeneous type $A_1 \to \cdots \to A_n \to o$. For this we take the index k such that $order(A_1) = \cdots = order(A_k) > order(A_{k+1})$. A term of the above type is safe if it is either: a constant; a variable; of a form $\lambda x_1^{A_1} \dots \lambda x_k^{A_k}.N$ with N a safe term whose all free variables have order bigger than $order(A_1)$; or of a form $MN_1 \dots N_k$ with $M, N_1, \dots, N_k$ safe terms of respective types.

The main point about safe terms is that variables need not be renamed during a reduction of a safe term. In a β-reduction step, $(\lambda x.M)N \to_\beta M[N/x]$, the definition of the substitution requires that free variables of N are not captured by binders in M. So, if $\lambda y.K$ is a subterm of M then either: (i) y should not be a free variable of N, or (ii) x should not appear free in K. Otherwise we need to rename y. If M is a safe term then by definition all free variables in K have order bigger than the order of y. So if x is free in K then $order(x) > order(y)$. Now since N is safe, the order of a free variable z of N is at least $order(N) = order(x)$. So $order(y) < order(x) \leq order(z)$ shows that $z \neq y$, hence we do not need to rename y. Thus the main point about the definition of safety is that the order of free variables in a safe term must be at least as big as the order of a term itself. The other conditions of the definition ensure that this property is preserved by β-reduction.

We will not give the definition of safe terms for λY-calculus [Salvati and Walukiewicz 2016]. It is quite similar but requires to make a distinction between variables bound by λ and those bound by Y. This is necessary to have an equivalence with safe recursive schemes as defined by Kanpik et al.[Knapik et al. 2002].

For safe terms we can use the same tools as in the previous subsection. First, we can reduce all the redexes of the highest order, say n. In the obtained term all redexes will be of order at most $n-1$. Moreover thanks to safety no renaming would be needed. Then we repeat reducing the redexes of the highest remaining order till none is left. So the complete transduction is now something like $smpl_1(link_1(\dots(smpl_n(link_n(tree(M)\dots)$ with $link_i$, $smpl_i$ working with variables of order i. This procedure works only because no step introduces new variable names. So if we start with a term from $Terms(\Sigma, \mathcal{T}, X)$ then after every $smpl_i(link_i(\cdot))$ we get again a term from $Terms(\Sigma, \mathcal{T}, X)$.

This way we obtain the transfer theorem for evaluation of safe terms. The exact formulation is as that of Theorem 2.2 but instead of restricting to terms with variables of type o, it requires that terms are safe.

2.5. Transfer for the full λY-calculus

For pushdown automata, as well as for terms with variables of type o, transductions and unfolding gave intuitively convincing ways to obtain the Böhm tree of a term. For safe terms the same approach worked when applied inductively on the order of redexes appearing in a term. For the full λY-calculus this does not work. The question what is the Böhm tree of a term is central to higher-order verification. Unlike for safe terms, we do not know simple basic operations with which we can construct Böhm trees of all terms of the λY-calculus. All the approaches mentioned below construct such a tree in one go.

Before we start, let us outline the tree automata approach to model-checking. Coupled with a good description of Böhm trees, it will give us decidability of higher-order model checking. The first step is to translate a property to be verified to a tree automaton. For our purposes let us take automata working on Σ-trees:

$$\mathcal{A} = \langle Q, \Sigma, q_{init} \in Q, \delta, \Omega : Q \to \mathbb{N} \rangle$$

where Q is a finite set of states, Σ is a tree signature, Ω is a parity acceptance condition, and δ is a transition function such that $\delta(q, b) \in \mathcal{P}(Q^k)$ where k is the number of arguments of b. So, for example, if the type of b is $o \to o \to o$ then $\delta(q, b) \in \mathcal{P}(Q \times Q)$. For a constant $c \in \Sigma$ of the base type o, we have $\delta(q, c) \in \mathcal{P}(Q^0) = \{f\!f, tt\}$; constant c labels a leaf in a tree, so $\delta(q,c)$ indicates if a run can reach the state q in a leaf labeled c. The next step is to take a transition system $\mathcal{S}$ to model check and to construct a two player game $\mathcal{K}(\mathcal{A},\mathcal{S})$. The idea is that this game characterizes when $\mathcal{A}$ accepts the unfolding of $\mathcal{S}$. The positions of the game are either pairs (q,v), where q is a state of $\mathcal{A}$ and v is a node of $\mathcal{S}$, or tuples $(q_1,\dots,q_k,v)$ if v is labeled by a letter of arity k. In a position of the form (q,v) a player, that we call Eve, chooses any position $(q_1,\dots,q_k)\in\delta(q,b)$ where b is the label of v. In this new position the other player, that we call Adam, chooses $i\in\{1,\dots,k\}$ so that the game can proceed to (v_i,q_i), where v_i is the i-th successor of v. A play in this game is an infinite sequence of moves, or a finite one if there is no outgoing move from some position. The later can happen only when v is labeled by a nullary constant, and then Eve wins if $\delta(q,v)=tt$. An infinite play is won by Eve if the sequence of states seen on the play is winning with respect to the parity condition Ω of the automaton. This way model-checking is reduced to game solving: the property given by $\mathcal{A}$ holds in a node v of $\mathcal{S}$ iff Eve has a winning strategy in $\mathcal{K}(A,\mathcal{S})$ from the position (q_{init},v).

To apply the outlined automata theoretic method one needs a suitable description of what a Böhm tree of a term is, since it will be $\mathcal{S}$ in the approach outlined above. First such description using game semantics was given by Ong [Ong 2006], and it him allowed to prove the decidability result. Another one is via an extension of higher-order pushdown automata with a collapse operation (CPDA) [Hague et al. 2008]. This split the decidability proof in two independent steps: (i) an equivalence of λY-calculus and CPDA; (ii) a decidability result for trees generated by CPDA. None of these steps is easy. Step (i) is a quite intricate challenge in programming of CPDA, and it has been the subject of independent studies [Carayol and Serre 2012; Salvati and Walukiewicz 2016]. Step (ii) is not easy either, it resembles a bit methods used for pushdown automata but requires to put CPDA into some normal form first.

Another approach came from the word of λ-calculus, where Krivine machines [Krivine 2007] have been used to understand β-reduction. Taking Krivine machines instead of CPDA greatly simplifies step (i) since equivalence of λY-calculus and trees generated by Krivine machines is almost evident. The good news is that this also simplifies step (ii) since Krivine machines have much more structure than CPDA, in particular they carry typing information from the term.

Yet another formalism has been proposed recently [Clemente et al. 2015]. Is is based on prefix tree rewriting and is more flexible in the sense that there are direct translations of Krivine machines and CPDA to this formalism. This flexibilty allowed for example to notice that verification of ordered pushdown systems [Atig 2012] is a particular instance of higher-order model checking.

Below we briefly outline the Krivine machine approach as it is the one that is most on the borderline between different communities.

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

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

As in this grammar, we will use $\emptyset$ for the empty environment. The notation $\rho[x\mapsto C]$ represents the environment that is as ρ but for the variable x where its value is C. Intuitively, a closure (M,ρ) denotes a closed λ-term: it is obtained by substituting for every free variable x of M the λ-term denoted by the closure $\rho(x)$.

A *configuration* of a 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

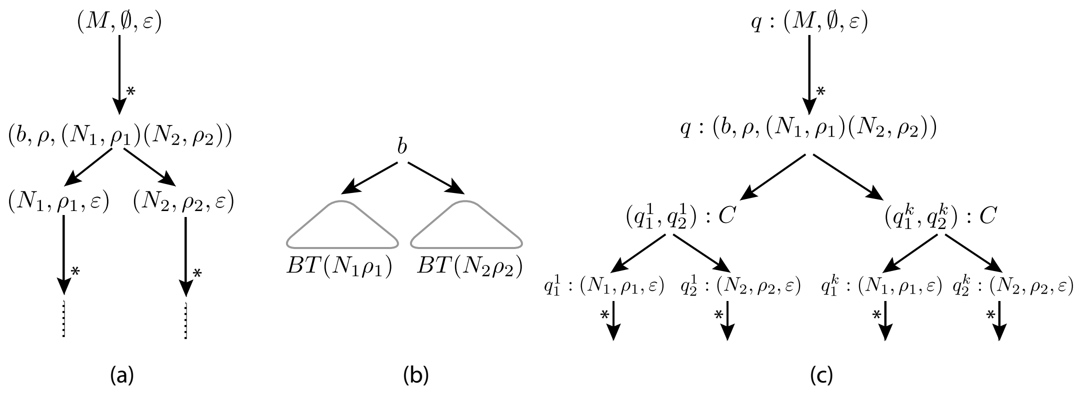

$(M, \emptyset, \varepsilon)$
$*$
$(b, \rho, (N_1, \rho_1)(N_2, \rho_2))$
$(N_1, \rho_1, \varepsilon)$
$(N_2, \rho_2, \varepsilon)$
$*$
$*$
(a)
b
$BT(N_1\rho_1)$
$BT(N_2\rho_2)$
(b)
$q : (M, \emptyset, \varepsilon)$
$*$
$q : (b, \rho, (N_1, \rho_1)(N_2, \rho_2))$
$(q_1^1, q_2^1) : C$
$(q_1^k, q_2^k) : C$
$q_1^1 : (N_1, \rho_1, \varepsilon)$
$q_2^1 : (N_2, \rho_2, \varepsilon)$
$q_1^k : (N_1, \rho_1, \varepsilon)$
$q_2^k : (N_2, \rho_2, \varepsilon)$
$*$
$*$
$*$
$*$
(c)

Fig. 7. Computation of the Krivine machine on $(M, \emptyset, \varepsilon)$, (a); from which we can read out the Böhm tree $BT(M)$, (b); and on which we can define the acceptance game $\mathcal{K}(\mathcal{A}, M)$, (c). Letter C abbreviates $(b, \rho, (N_1, \rho_1)(N_2, \rho_2))$.

topmost element of the stack is on the left. The empty stack is denoted by ε. The rules of a Krivine machine are as follows:

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

The first rule says that in order to evaluate an abstraction $\lambda x.M$, we should look for the argument at the top of the stack, then we bind this argument to x, and calculate the value of M. To evaluate an application MN we create a closure out of N and the current environment so as to be able to evaluate N correctly when necessary and put that closure on the stack; then we continue to evaluate M. The rule for $Yx.M$ simply amounts to bind the variable x in the environment to the current closure of $Yx.M$ and to calculate M. Finally, the rule for variables says that we should take the value of the variable from the environment and should evaluate it; the value is not just a term but a closure: a term with an environment giving the right meanings to the free variables of the term.

Note that the rules of a Krivine machine are deterministic and that there is one rule per form of a term. The rules preserve typing in a sense that every configuration reachable from $(M, \emptyset, \varepsilon)$ with M of type o is of the form (N, ρ, S) where (i) N is a subterm of M; (ii) ρ is defined for all free variables in N and assigns to them closures of appropriate types, and (iii) S is a sequence $C_1 \dots C_k$ of closures of types $A_1, \dots, A_k$, respectively, where $A_1 \to \dots A_k \to o$ is the type of N. In particular, the length of S is bounded.

Let us explain how to use Krivine machines to calculate the Böhm tree of a term, cf. Figure 7(a). For our purposes it is enough to consider a closed term M of the base type. We start the machine in a configuration with the empty environment and stack: $(M, \emptyset, \varepsilon)$. The machine is deterministic, but it may not stop due to the fixpoint rule. If it does not stop then $BT(M) = \Omega^0$. If it does stop then, since its execution preserves typing, the reached configuration must be of the form $(b, \rho, (N_1, \rho_1)(N_2, \rho_2))$, for simplicity suppose that the arity of b is 2. The type of b is $o^2 \to o$; and N_1, N_2 are terms of type o. In this case b is the root of $BT(M)$ and the subtrees attached to the root are Böhm trees computed form (N_1, ρ_1), and (N_2, ρ_2), cf. Figure 7(b).

The next step is exactly the same as in the standard automata theoretic approach outlined above. We take the computation tree of the Krivine machine $\mathcal{K}(M)$, as in

Figure 7(a), make a product with a tree automaton $\mathcal{A}$ recognizing the property. This gives the a game $\mathcal{K}(\mathcal{A}, M)$ from Figure 7(c). There is a new branching corresponding to the choice of a transition of the automaton by Eve. So the question whether $BT(M)$ is accepted by $\mathcal{A}$ is reduced to the question of existence of a winning strategy for Eve in $\mathcal{K}(\mathcal{A}, M)$.

The game $\mathcal{K}(\mathcal{A}, M)$ is infinite because $\mathcal{K}(M)$ is infinite due to environments that may grow arbitrary big. One can think of a Krivine machine as a kind of pushdown machine with a tree-shaped stack. not to be confused with the stack of the Krivine machine itself

The last step is to construct ~~form~~ $\mathcal{K}(\mathcal{A}, M)$ a finite game $\mathcal{G}(\mathcal{A}, M)$. The idea is to replace closures by elements from a finite domain. This domain will depend on the ranks used in the acceptance condition of $\mathcal{A}$, say that the range of Ω is $\{1, \dots, d\}$. We let $\mathcal{R}_o$, the set of *residuals* of type o, to be $\mathcal{P}(Q) \times \{1, \dots, d\}$; and $\mathcal{R}_{A \to B}$ to be the set of functions from A to B. The positions of $\mathcal{G}(\mathcal{A}, M)$ are of the form $q : (N, \rho^{\mathcal{R}}, S^{\mathcal{R}})$ where N is a subterm of M, $\rho^{\mathcal{R}}$ assigns residuals of appropriate types to variables, and $S^{\mathcal{R}}$ is a stack of residuals of types determined by the type of N. So the number of positions is finite. It can be then shown that $\mathcal{G}(\mathcal{A}, M)$ is equivalent to $\mathcal{K}(\mathcal{A}, M)$ in a sense that the winner in the two games is the same [Salvati and Walukiewicz 2014]. Game $\mathcal{G}(\mathcal{A}, M)$ can be then solved algorithmically since it is finite.

Finally, one can observe that there is an MSOL-transduction defining $\mathcal{G}(\mathcal{A}, M)$ from $tree(M)$ provided M comes from some fixed set $Terms(\Sigma, \mathcal{X}, \mathcal{T})$ for some finite set of variables $\mathcal{X}$, and a finite set of types $\mathcal{T}$. This construction gives the desired result [Salvati and Walukiewicz 2013]

THEOREM 2.3 (TRANSFER THEOREM). *Fix Σ, $\mathcal{T}$, and $\mathcal{X}$. For every MSOL formula φ one can effectively construct an MSOL formula $\widehat{\varphi}$ such that for every closed λY-term $M \in Terms(\Sigma, \mathcal{T}, \mathcal{X})$ of type o:*

$$M \vDash \widehat{\varphi} \qquad \textit{iff} \qquad BT(M) \vDash \varphi.$$

The transfer theorem can be also derived from the typing system of Kobayashi and Ong [Kobayashi and Ong 2009]. In [Salvati and Walukiewicz 2013] Transfer Theorem is also proved for infinite λ-terms. The restriction to $Terms(\Sigma, \mathcal{T}, \mathcal{X})$ looks to be unavoidable: the transfer property does not hold for the class of all terms if the polynomial time hierarchy is strict [Salvati and Walukiewicz 2013].

3. RECOGNIZABILITY

Recognizability by a morphism from Σ^* to a finite semi-group is a standard notion from language theory [Eilenberg 1974]. Recognizable languages of finite words are precisely regular languages, or equivalently, languages definable in MSOL. Here we will take the same definition but instead of semi-groups we take models for the λY-calculus. We will see that in this case we can also recognize all MSOL definable languages of λY-terms. But, at present it is not known if there are some other languages of λY-term that are recognizable.

A *model* $\mathcal{D}$ is a family of sets indexed by types, $\{D_A\}_{A \in \mathcal{T}}$, together with an interpretation $[\![\cdot,\cdot]\!]^{\mathcal{D}}$ mapping a term M of type A and a valuation v to an element $[\![M, v]\!]$ of D_A. As usual, a valuation is a function assigning elements of the model to variables; this function should respect typing, meaning that $v(x^A) \in D_A$ for every variable x^A. Recall that variables are explicitly typed but we often do not write this type when it is not important. Similarly, we will often omit the superscript $\mathcal{D}$ in $[\![\cdot]\!]^{\mathcal{D}}$. While there are more general formulations, here we will simply require that $D_{A \to B}$ is a subset of $D_A \to D_B$, so elements of a higher type are functions on elements of a lower type. A model must satisfy the following identities:

$$
\begin{aligned}
[\![x, v]\!] &= v(x)\\
[\![MN, v]\!] &= [\![M, v]\!]([\![N, v]\!])\\
[\![\lambda x^A.M, v]\!](d) &= [\![M, v[d/x]]\!] \qquad \text{for every } d \in D_A\\
[\![Y^{(A\to A)\to A}, v]\!](d) &= d([\![Y, v]\!](d)) \qquad \text{for every } d \in D_A
\end{aligned}
$$

These identities guarantee that $[\![M, v]\!] \in D_A$ for every term of type A. They imply also that a model assigns the same value to $\beta\delta$-equal terms, i.e., if $M \to_{\beta,\delta} N$ then $[\![M, v]\!] = [\![N, v]\!]$, for arbitrary v. If M is a closed term, we will simply write $[\![M]\!]$ without mentioning a valuation.

A model $\mathcal{D}$ can be used to *recognize* a set of terms [Salvati 2009]. Given a set of elements of the base type $F \subseteq D_o$, the *language recognized by F in $\mathcal{D}$* is a set of closed terms of the base type: $\{M : [\![M]\!]^{\mathcal{D}} \in F\}$.

This definition of recognizability is of the same form as recognizability of word languages by semi-groups. The nice thing is that for words the two are equivalent in the following sense. Recall from page 17 that a word w can be represented as a term M_w of type $o \to o$. Let us suppose that we have a constant $c : o$ just to be able to construct terms of type o; so $M_w c$ has type o. For every finite model $\mathcal{D}$ and $F \in \mathcal{D}_o$ the set $\{w : [\![M_w c]\!] \in F\}$ is recognizable by a finite semi-group; and conversely, for every set of words recognizable by a finite semi-gorup we can find an appropriate model $\mathcal{D}$ and $F \in \mathcal{D}_o$.

By definition, the set of recognizable sets of terms is closed under union, intersection, and complement. The important difference with recognizability for words is that it is not closed under relabeling [Salvati 2009], in particular a problem arises when two constants are relabeled to the same constant. Another important difference is that the emptiness of a recognizable set is not decidable: one cannot determine if there is a term of a given value. This is a reformulation of the result of Loader about undecidability of λ-definability [Loader 2001].

The definition of the model allows to interpret Y as an arbitrary fixpoint. At present we do not know what is the recognition power of such models, the difficulty comes from the fact that it is not clear how to handle arbitrary fixpoints. We know the answer in some particular cases though.

The first obvious thing to consider are greatest fixpoints. A *GFP model* is when D_o is a finite lattice, and $D_{A\to B}$ is the set of monotone functions in $D_A \to D_B$ ordered componentwise. As the name suggests, Y is interpreted as the greatest fixpoint, and Ω^A as the greatest element $\top_A$ in D_A. By inverting the order we also obtain LFP models.

As an example of a GFP model, consider $\mathcal{D}_2$ with $D_o = \{\bot, \top\}$. All constants other than Y and Ω are interpreted as the least element of the appropriate type. This model can recognize terms that do not have a head normal form:

$$BT(M) = \Omega \quad \text{iff} \quad [\![M]\!]^{\mathcal{D}_2} = \top.$$

The next theorem characterizes the recognition power of GFP-models in terms of tree automata. A tree automaton as defined on page 20 is a *TAC-automaton* if $\Omega(q) = 0$ for all states of the automaton. TAC stands for trivial acceptance condition, as indeed every infinite play in the acceptance game is winning for Eve. TAC automata express exactly the properties expressed in the mu-calculus with formulas using solely the greatest fixpoint operator (and no negation). An automaton is moreover *Ω-blind* if $\delta(q, \Omega) = tt$ for all states q; this means that it cannot test for divergence.

THEOREM 3.1. *[[Aehlig 2007],[Salvati and Walukiewicz 2015c]] A language L of λY-terms is recognized by a GFP-model iff it is a boolean combination of languages of Ω-blind TAC automata.*

We would like to outline how to construct a model for a given TAC automaton. This will, almost, prove the right-to-left implication of the theorem. Given an automaton $\mathcal{A}$ we construct a model $\mathcal{D}^{\mathcal{A}}$. The elements of the base set D_o are just the subsets of the set of states of $\mathcal{A}$: $D_o^{\mathcal{A}} = \mathcal{P}(Q)$. This determines D_A for every type A. A constant c of type o is interpreted as the set $\{q : \delta(q,c) = tt\}$. A constant b of type $0^k \to 0$ is interpreted as the function whose value on $(S_1,\dots,S_k) \in \mathcal{P}(Q)^k$ is $\{q : \delta(q,b) \cap S_1 \times \cdots \times S_k \neq \emptyset\}$. Finally, for the set $F^{\mathcal{A}}$ used to recognize $L(\mathcal{A})$ we put $S \in F^{\mathcal{A}}$ iff S contains the initial state of $\mathcal{A}$.

We want to show that for every closed term M of type o:

$$BT(M) \in L(\mathcal{A}) \quad \text{iff} \quad [\![M]\!]^{\mathcal{D}^{\mathcal{A}}} \in F^{\mathcal{A}}. \tag{1}$$

A nice property of GFP-models is that we can define the semantics of a Böhm tree of a term using its truncations. For every $n \in \mathbb{N}$, we denote by $BT(M)\!\downarrow_n$ the finite term that is the result of replacing in the tree $BT(M)$ every subtree at depth n by the constant Ω^A of the appropriate type. Observe that if M is closed and of type o then A will be o too. This is because we work with a tree signature. We define

$$[\![BT(M), v]\!] = \bigwedge\{[\![BT(M)\!\downarrow_n, v]\!] : n \in \mathbb{N}\}. \tag{2}$$

An important consequence of this definition is that $[\![BT(M)]\!] = [\![M]\!]$ for every closed term M (c.f. [Amadio and Curien 1998] Exercise 6.1.8).

Coming, back to the eqivalence (1), we first prove the left-to-right direction. We take a λY-term M such that $BT(M) \in L(\mathcal{A})$, and show that the initial state q^0 is in $[\![BT(M)]\!]$. This will do as $[\![BT(M)]\!] = [\![M]\!]$ by the above paragraph. Thanks to equation (2), it is enough to show that $q^0 \in [\![BT(M)\!\downarrow_n]\!]$ for every n supposing that $\mathcal{A}$ accepts $BT(M)$. This is proved by examining the definition of the model.

To prove the right-to-left direction of equivalence (1), we take a term M and a state $q \in [\![M]\!] = [\![BT(M)]\!]$. We show that $\mathcal{A}$ accepts $BT(M)$ from q. For this we say how Eve should play to win in the acceptance game. For example, if $BT(M)$ is of the form $bN_1 \dots N_k$ then by the definition of $[\![b]\!]$, there is $(q_1,\dots,q_k)$ in $\delta(q,b)$ so that $q_i \in [\![N_i]\!]$, for all $i \in \{1,\dots,k\}$. Thus Eve can choose this transition, then Adam chooses a direction i, and then Eve can proceed with q_i from the i-th child of the root. This strategy ensures that Eve does not loose a finite play. Since $\mathcal{A}$ is a TAC automaton, every infinite play is winning for Eve. So the strategy is winning for Eve, and $\mathcal{A}$ accepts $BT(M)$ from q.

As the construction of the model $\mathcal{D}^{\mathcal{A}}$ is effective, property (1) gives decidability of higher-order model checking for properties expressed by TAC automata. It is is enough to take a term and evaluate it in the model using the identities above. The value of a term of type o will be a set of states of the automaton. The property holds if the initial state is in this set.

This is a strikingly simple decidability proof. We have outlined one direction of the proof of Theorem 3.1, the other is not more complicated, moreover the whole proof uses only standard techniques from λ-calculus. It is then very tempting to try to construct models for more complicated automata. The left-to-right direction of Theorem 3.1 predicts some difficulties though as more complicated automata will need more complicated fixpoints.

The first generalization of Theorem 3.1 is to consider consider all TAC automata, and not necessarily Ω-blind ones. The idea is to combine model $\mathcal{D}_2$ for recognizing Ω with a GFP model for a blind TAC automaton. Intuitively, $\mathcal{D}_2$ model changes Ω to a "normal" label and then the blind TAC automaton can read this label without restrictions [Salvati and Walukiewicz 2015c].

The idea of combining models was then streamlined and generalized in [Salvati and Walukiewicz 2015b] to obtain the result for weak MSOL. Automata recognizing weak MSOL have a special form. One can think of such an automaton as working in phases: first runs an Ω-blind TAC automaton and labels the tree, then a dual to an Ω-blind TAC automaton reads this labeling and produces a new one, then it is the turn of another Ω-blind TAC automaton, and so on for a fixed number of phases. In semi-group theory such a cascade composition of automata is captured by the wreath product of semi-groups recognizing languages of respective automata. In [Salvati and Walukiewicz 2015b] we do a loosely similar construction but only for a very specific class of models that is sufficient for weak MSOL.

For all MSOL properties the construction is more complicated [Salvati and Walukiewicz 2015a]. We have no handy decomposition principle, so we need to construct the model in one step.

THEOREM 3.2 (RECOGNIZABILITY). *For every MSOL formula φ there is a model recognizing the set of terms M such that $BT(M) \vDash \varphi$.*

The type system Tsukada and Ong [Tsukada and Ong 2014] can also be used to obtain such a model, although at the moment there is no explicit description of the semantics of the fixpoint operator. Yet another approach has been proposed by Grellois and Melliès [Grellois and Melliès 2015; Grellois 2016] who derive a model by extending some constructions of models for linear logic.

The model construction implies the transfer theorem. Fixing Σ, $\mathcal{T}$, and $\mathcal{X}$ as in the transfer theorem; for every finitary model, and every element d of the model there is an MSOL formula that holds precisely in terms $M \in \mathit{Terms}(\mathcal{S}, \mathcal{T}, \mathcal{X})$ whose value in the model is d. The formula simply guesses a value for each subterm and checks that these guesses are consistent with respect to the definition of application and abstraction in the model.

A finitary model gives also a procedure for some kind of synthesis of programs from modules. Given a formula φ and a finite set of closed λY-terms $M_1 \dots, M_l$, it is decidable if there is a closed term K constructed from $M_1 \dots, M_l$ by means of application, Y-variables and Y-binders, such that $BT(K) \vDash \varphi$. If there is such a term then there exists a finite one. The idea is to construct a finite automaton accepting such terms K. The transitions of the automaton are determined by meanings of $M_1 \dots, M_l$ in the model. Every, infinite, tree accepted by this automaton gives a desired, infinite, term K. Automata theory tells us that if an automaton accepts a tree then it also accepts some regular tree. This regular tree can be represented as a finite term with a help of fixpoint operators.

4. CONCLUSIONS

We have presented some results and methods used in higher-order model-checking. These came from automata theory, logic, semantics, and lambda-calculus. In this overview we have approached the subject from automata theoretic perspective. A more linear logic oriented perspective can be found in the recent PhD thesis of Grellois [Grellois 2016]. A more typing oriented view is presented in a survey of Ong [Ong 2015].

The decidability of higher-order model-checking [Ong 2006] is clearly the cornerstone of the subject. We have chosen to present two more general results: the transfer theorem, and recognizability. The first follows the automata theoretic tradition. For safe terms one can prove the result by induction on the order of types using some transductions and unfoldings. For all terms, the approaches we know show the result in one step. It is not clear if it can be decomposed into more basic ones. The notion of recognizability of λY-terms raises new semantical questions motivated by automata theory. It requires to study models with fixpoints that are neither least nor greatest fixpoints. At present we do not have a semantical characterization of those fixpoints that are sufficient to recognize all properties expressible in monadic second-order logic.

We have said nothing about applications of higher-order model checking to program verification. As we have seen, higher-order model-checking is conceptually much simpler when restricted to properties expressed by tree automata with trivial acceptance conditions (TAC automata) [Aehlig 2007]. Kobayashi proposed a type system for such properties and constructed a tool based on it [Kobayashi 2013]. This in turn opened the way to an active ongoing research resulting in the steady improvement of the capacities of the verification tools [Broadbent et al. 2013; Broadbent and Kobayashi 2013; Ramsay et al. 2014; Murase et al. 2016]. Despite non-elementary complexity lower bounds for the higher-order model-checking problem [Engelfriet 1983; Kobayashi and Ong 2011] current tools can analyze programs of several thousands of lines. This success is another driving force for the subject.

REFERENCES

Klaus Aehlig. 2007. A Finite Semantics of Simply-Typed Lambda Terms for Infinite Runs of Automata. *Logical Methods in Computer Science* 3, 1 (2007), 1–23.

Roberto M. Amadio and Pierre-Louis Curien. 1998. *Domains and Lambda-Calculi*. Cambridge Tracts in Theoretical Computer Science, Vol. 46. Cambridge University Press.

Mohamed F. Atig. 2012. Model-Checking of Ordered Multi-Pushdown Automata. *Log. Methods Comput. Sci.* 8, 3 (09 2012). DOI:http://dx.doi.org/10.2168/LMCS-8(3:20)2012

W. Blum. 2009. *The safe lambda calculus*. Ph.D. Dissertation. Oxford University.

William Blum and C.-H. Luke Ong. 2009. The Safe Lambda Calculus. *Logical Methods in Computer Science* 5, 1 (2009).

Achim Blumensath, Thomas Colcombet, and Christof Löding. 2008. Logical theories and compatible operations. In *Logic and Automata (Texts in Logic and Games)*, Jörg Flum, Erich Grädel, and Thomas Wilke (Eds.), Vol. 2. Amsterdam University Press, 73–106.

Christopher H. Broadbent, Arnaud Carayol, Matthew Hague, and Olivier Serre. 2013. C-SHORe: a collapsible approach to higher-order verification. In *ICFP*. ACM, 13–24.

Christopher H. Broadbent and Naoki Kobayashi. 2013. Saturation-Based Model Checking of Higher-Order Recursion Schemes. In *CSL (LIPIcs)*, Vol. 23. Schloss Dagstuhl, 129–148.

J. R. Büchi. 1960. Weak Second-Order Arithmetic and Finite Automata. *Z. Math. Logik Grundl. Math.* 6 (1960), 66–92.

Arnaud Carayol and Olivier Serre. 2012. Collapsible Pushdown Automata and Labeled Recursion Schemes Equivalence, Safety and Effective Selection. In *LICS*. 165–174.

Lorenzo Clemente, Pawel Parys, Sylvain Salvati, and Igor Walukiewicz. 2015. Ordered Tree-Pushdown Systems. In *FSTTCS'15 (LIPIcs)*, Vol. 45. 163–177. DOI:http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2015.163

Bruno Courcelle. 1978. A Representation of Trees by Languages I. *Theor. Comput. Sci.* 6 (1978), 255–279.

Bruno Courcelle. 1994. Monadic Second-Order Graph Transductions: A Survey. *Theoretical Computer Science* 126 (1994), 53–75.

Bruno Courcelle. 1995. The Monadic Second-Order Logic on Graphs IX: machines and behaviours. *Theoretical Computer Science* 149 (1995).

Bruno Courcelle and Teodor Knapik. 2002. The evaluation of first-order substitution is monadic second-order compatible. *Theoretical Computer Science* 281 (2002), 177–206. Special issue offered to Maurice Nivat.

Bruno Courcelle and Igor Walukiewicz. 1998. Monadic Second-Order Logic, Graphs and Unfoldings of Transition Systems. *Annals of Pure and Applied Logic* 92 (1998), 35–62.

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

S. Eilenberg. 1974. *Automata, Languages and Machines*. Vol. A. Academic Press, New York.

Calvin C. Elgot. 1971. Algebraic theories and program schemes. See Engeler [1991], 71–88. DOI:http://dx.doi.org/10.1007/BFb0059694

Erwin Engeler (Ed.). 1991. *Symposium on Semantics of Algorithmic Languages*. Lecture Notes in Mathematics, Vol. 188. Springer. DOI:http://dx.doi.org/10.1007/BFb0059689

J. Engelfriet. 1983. Iterated Push-Down Automata and Complexity Classes. In *15th STOC'83*. 365–373.

J. Engelfriet and E.M. Schmidt. 1977. IO and OI. *J. Comput. System Sci.* 15, 3 (1977), 328–353.

J. Engelfriet and E.M. Schmidt. 1978. IO and OI. II. *Journal of computer and system sciences* 16 (1978), 67–99.

Charles Grellois. 2016. *Semantics of linear logic and higher-order model-checking. (Sémantique de la logique linéaire et "model-checking" d'ordre supérieur)*. Ph.D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-01311150

Charles Grellois and Paul-André Melliès. 2015. Finitary Semantics of Linear Logic and Higher-Order Model-Checking. In *MFCS'15 (LNCS)*, Vol. 9234. 256–268. DOI:http://dx.doi.org/10.1007/978-3-662-48057-1_20

Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. 2008. Collapsible Pushdown Automata and Recursion Schemes. In *LICS'08*. IEEE Computer Society, 452–461.

Y. Ianov. 1960. The logical schemes of algorithms. In *Problems of Cybernetics I*. Pergamon, Oxford, 82–140.

Klaus Indermark. 1976. Schemes with Recursion on Higher Types. In *MFCS'76 (LNCS)*, Vol. 45. 352–358.

Petr Jancar. 2012. Decidability of DPDA Language Equivalence via First-Order Grammars. In *LICS'12*.

T. Knapik, D. Niwinski, and P. Urzyczyn. 2002. Higher-order pushdown trees are easy. In *FoSSaCS' 02 (LNCS)*, Vol. 2303. 205–222.

N. Kobayashi. 2013. Model Checking Higher Order Programs. *J. ACM* 60 (2013).

Naoki Kobayashi and C.-H. Luke Ong. 2011. Complexity of Model Checking Recursion Schemes for Fragments of the Modal Mu-Calculus. *Logical Methods in Computer Science* 7, 4 (2011).

N. Kobayashi and L. Ong. 2009. A Type System Equivalent to Modal Mu-Calculus Model Checking of Recursion Schemes. In *LICS'09*. 179–188.

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

R. Loader. 2001. The Undecidability of lambda-definability. In *Logic, Meaning and Computation: Essays in memory of Alonzo Church*, C. A. Anderson and M. Zeleny (Eds.). Kluwer, 331–342.

Robin Milner. 1973. *Models of LCF*. Memo AIM-186. Stanford University.

D.E. Muller and P.E. Schupp. 1985. The Theory of Ends, Pushdown Automata and Second-Order Logic. *Theoretical Computer Science* 37 (1985), 51–75.

A. Murase, T. Terauchi, N. Kobayashi, R. Sato, and H. Unno. 2016. Temporal Verification of Higher-Order Functional Programs. In *POPL*. 57–68.

M. Nivat. 1975. On interpretation of polyadic recursive program schemes. *Symposia Mathematica* 15 (1975), 255–281.

C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *LICS'06*. 81–90.

Luke Ong. 2015. Higher-Order Model Checking: An Overview. In *LICS*. 1–15. DOI:http://dx.doi.org/10.1109/LICS.2015.9

Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255.

Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. 2014. A type-directed abstraction refinement approach to higher-order model checking. In *POPL*. ACM, 61–72.

Sylvain Salvati. 2009. Recognizability in the Simply Typed Lambda-Calculus. In *WoLLIC'09 (LNCS)*, Vol. 5514. 48–60.

Sylvain Salvati and Igor Walukiewicz. 2013. Evaluation is MSOL-compatible. In *FSTCS (LIPIcs)*, Vol. 24. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 103–114. DOI:http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2013.103

Sylvain Salvati and Igor Walukiewicz. 2014. Krivine machines and higher-order schemes. *Inf. Comput.* 239 (2014), 340–355. DOI:http://dx.doi.org/10.1016/j.ic.2014.07.012

Sylvain Salvati and Igor Walukiewicz. 2015a. A Model for Behavioural Properties of Higher-order Programs. In *CSL'15 (LIPIcs)*, Vol. 41. 229–243. DOI:http://dx.doi.org/10.4230/LIPIcs.CSL.2015.229

Sylvain Salvati and Igor Walukiewicz. 2015b. Typing Weak MSOL Properties. In *FOSSACS 15*, Vol. 9034. 343–357. DOI:http://dx.doi.org/10.1007/978-3-662-46678-0_22

Sylvain Salvati and Igor Walukiewicz. 2015c. Using models to model-check recursive schemes. *Logical Methods in Computer Science* 11, 2 (2015). DOI:http://dx.doi.org/10.2168/LMCS-11(2:7)2015

Sylvain Salvati and Igor Walukiewicz. 2016. Simply typed fixpoint calculus and collapsible pushdown automata. *Mathematical Structures in Computer Science* FirstView (4 2016), 1–47. DOI:http://dx.doi.org/10.1017/S0960129514000590

D. Scott. 1972. Continuous Lattices. In *Proc. of Dalhousie Conference (Lecture Notes in Mathematics)*, Vol. 188. Springer, 311–366.

Dana S. Scott. 1971. The lattice of flow diagrams. See Engeler [1991], 311–366. DOI:http://dx.doi.org/10.1007/BFb0059703

A.L. Semenov. 1984. Decidability of Monadic Theories. In *MFCS'84 (LNCS)*, Vol. 176. Springer-Verlag, 162–175.

Géraud Sénizergues. 2001. L(A)=L(B)? Decidability results from complete formal systems. *Theor. Comput. Sci.* 251, 1-2 (2001), 1–166.

Géraud Sénizergues. 2002. L(A)=L(B)? A simplified decidability proof. *Theor. Comput. Sci.* 281, 1-2 (2002), 555–608.

Colin Stirling. 2002. Deciding DPDA Equivalence Is Primitive Recursive. In *ICALP'02 (LNCS)*, Vol. 2380. 821–832.

Takeshi Tsukada and C.-H. Luke Ong. 2014. Compositional higher-order model checking via ω-regular games over Böhm trees. In *LICS-CSL*. 78:1–78:10. DOI:http://dx.doi.org/10.1145/2603088.2603133

I. Walukiewicz. 2002. Monadic Second Order Logic on Tree-Like Structures. *Theoretical Computer Science* 275 (2002), 311–346.