The Monadic Second Order Theory of Trees Given by Arbitrary Level-Two Recursion Schemes Is Decidable

Klaus Aehlig*, Jolie G. de Miranda, and C.-H. Luke Ong

Oxford University Computing Laboratory

Abstract. A tree automaton can simulate the successful runs of a word or tree automaton working on the word or tree denoted by a level-2 lambda-tree. In particular the monadic second order theory of trees given by arbitrary, rather than only by safe, recursion schemes of level 2 is decidable. This solves the level-2 case of an open problem by Knapik, Niwiński and Urzyczyn.

1 Introduction and Related Work

Since Rabin [11] showed the decidability of the monadic second order theory of the binary tree this result has been applied and extended to various mathematical structures, including algebraic trees [4] and a hierarchy of graphs [3] obtained by iterated unfolding and inverse rational mappings from finite graphs. The interest in these kinds of structures arose in recent years in the context of verification of infinite state systems [9, 13].

Recently Knapik, Niwiński and Urzyczyn [6] showed that the monadic second order (MSO) theory of any infinite tree generated by a level-2 grammar satisfying a certain "safety" condition is decidable. Later they generalised [7] this result to grammars of arbitrary levels, but still requiring the "safety" condition. It remains open whether this condition is actually needed. In this article we give a partial answer: For grammars of level 2 the condition can be dropped.

Two observations are essential to obtain the result. The first, albeit trivial, is that if we go down to level 0, we will never actually perform a substitution, thus we need not worry that substitution is capture avoiding. *If you don't do a substitution, you'll never do a wrong substitution!*

The second observation is that even though first-order variables stand for words or trees of unbounded lengths, all the information we need to know in order to check for a *particular property* is the transition function of the automaton verifying this property. And this is a bounded amount of information!

Therefore the run of a Büchi automaton on an ω-word can be simulated by a Büchi tree automaton on a second order lambda tree denoting this word. Moreover, this idea extends to the simulation of an alternating parity-tree automaton by a two-way alternating parity tree automaton. It follows that the full MSO theory of the tree language generated by a level-2 recursion scheme is decidable.

---

* On leave from Ludwig-Maximilians-Universität München. Supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD).

It should be mentioned that in another article [1] the authors show a related result. For *word languages* general level-2 recursion schemes and safe recursion schemes indeed produce the same set of languages and the transformation is effective. It is yet unclear whether that result extends to tree languages. Moreover, the authors believe that the conceptual simplicity of the method presented here makes it worth being studied in its own right.

Only after finishing the work presented here the authors became aware of a manuscript by Knapik, Niwiński, Urzyczyn and Walukiewicz [8] who also solved the level-2 decidability problem. They used a new kind of automaton equipped with a limited "backtracking" facility.

The article is organised as follows. In Sections 2 and 3 we introduce lambda trees and recursion schemes. Section 4 shows the expected connection between the denotation of a recursion scheme and that of the associated lambda tree. Section 5 explains the main technical idea of the article: if we are interested in a particular property, all we need to know about a first-order object can be described by a bounded amount of information. Sections 6 and 7 show how the idea can be used to obtain the decidability of the MSO theory of words and trees given by level-2 recursion schemes.

2 Lambda Trees

Since the main technical idea for deciding properties of recursion schemes is to translate them to properties of infinitary lambda terms [5] we first have to consider these terms *qua* trees. In this section we will only handle abstractions that (morally) handle first-order abstractions (even though we give an untyped definition). The extension to function abstractions is explained in Section 5.

We presuppose a countably infinite set $\mathcal{V}$ of variables x.

Definition 1 (lambda trees). A *lambda tree* is a, not necessarily well-founded, tree built from the binary constructor *application* @, and for every variable x a unary *abstraction* constructor λ_x, and nullary *variable* constructor v_x. Moreover there is an unspecified but finite set Σ of constants, called "letters".

A lambda tree defines a, potentially partial, ω-word in a natural way made precise by the following definitions. They are motivated by ideas of geometry of interaction [2] and similar to the ones presented by Knapik *et al* [6].

Definition 2 (the matching lambda of a variable). Let p be a node in a lambda tree that is a variable v_x. Its *matching path* is the shortest prefix (if it exists) of the path from p to the root of the tree that ends in a λ_x node. We call the last node of the matching path the *matching lambda* of the variable node p. If no such path exists, we say the variable node p is a *free variable* v_x. Variable nodes that are not free are called *bound*.

In this definition, if we replace "root of the tree" by a node r we get the notion of a variable *free in the (located) subterm* r.

Definition 3 (matching argument of a lambda or a letter). For a lambda tree we define the k-th argument of a node, which is assumed to be a lambda or a letter, to be the right-hand child of the application node (if it exists) where, when walking from the given node to the root, for the first time the number of applications visited exceeds the number of abstractions visited (not including the starting node) by k. We presuppose that on this path, called the *matching path*, application nodes are only visited from the left child. We define the matching argument of a λ-node to be its first argument.

Definition 4 (canonical traversal of a lambda-tree). The *canonical traversal* of a lambda tree starts at the root of the tree. From an application we go to the left subtree, from an abstraction we go to the body. From a letter we go to its first argument. From a variable we first go to the matching lambda and then from there to the matching argument where we continue as above.

If we collect on the canonical traversal all the letters we pass downwards (that is, in direction from the root to the leaves) in order of traversal we get a, maybe partially defined, ω-word. This word is called *the word met on the canonical traversal.* If, instead of always going to the first argument when we meet a letter we branch and continue with the i'th argument, for $1 \leq i \leq k$ where k is the "arity" of the letter (assuming a fixed assignment), we obtain a tree, *the tree of the canonical traversal.*

Proposition 5. *In the canonical traversal (and in every path of the canonical tree traversal) every node is visited at most three times. More precisely, each node is visited at most once from each direction (top, left and right child).*

Remark 6. It should be noted that all the notions and results in this section are invariant under renaming of bound variables in a lambda tree in the usual way. This will be used tacitly in the sequel.

3 Recursion Schemes

Definition 7 (simple types and their level). Given a base type ι, the *simple types over* ι are inductively defined as the smallest set containing ι that is closed under forming arrow types $\sigma \to \tau$. We use the expression "simple type" if ι is understood from the context or irrelevant. We understand that $\to$ associates to the right, so $\sigma \to \tau \to \tau'$ is short for $\sigma \to (\tau \to \tau')$.

The level $\mathrm{lv}(\tau)$ of a simple type is inductively defined by $\mathrm{lv}(\iota) = 0$ and $\mathrm{lv}(\sigma \to \tau) = \max(\mathrm{lv}(\sigma) + 1, \mathrm{lv}(\tau))$. We use the expression "type of level i" to mean types τ with $\mathrm{lv}(\tau) = i$.

Definition 8 (combinatory terms). Given a set $\mathcal{C}$ of typed constants and a set $\mathcal{V}$ of typed variables, the sets $\mathcal{T}^\tau(\mathcal{C}, \mathcal{V})$ of terms (over $\mathcal{C}$ and $\mathcal{V}$) of type τ is inductively defined as follows.

- $x \in \mathcal{T}^\tau(\mathcal{C}, \mathcal{V})$ if $x \in \mathcal{V}$ is of type τ and $C \in \mathcal{T}^\tau(\mathcal{C}, \mathcal{V})$ if $C \in \mathcal{C}$ is of type τ
- if $s \in \mathcal{T}^{\tau\to\sigma}(\mathcal{C}, \mathcal{V})$ and $t \in \mathcal{T}^\tau(\mathcal{C}, \mathcal{V})$ then $st \in \mathcal{T}^\sigma(\mathcal{C}, \mathcal{V})$

Proposition 9. *Every term in* $\mathcal{T}^{\tau}(\mathcal{C}, \mathcal{V})$ *is of one of the following forms.*

- $C\overrightarrow{t}$ *with* $C \in \mathcal{C}$ *of type* $\overrightarrow{\tau} \to \sigma$ *and* $t_i \in \mathcal{T}^{\tau_i}(\mathcal{C}, \mathcal{V})$
- $x\overrightarrow{t}$ *with* $x \in \mathcal{V}$ *of type* $\overrightarrow{\tau} \to \sigma$ *and* $t_i \in \mathcal{T}^{\tau_i}(\mathcal{C}, \mathcal{V})$

Definition 10 (recursion scheme). A level-2 *recursion scheme* is given by the following data.

- A finite set $\mathcal{N}$ of typed constants, called "non-terminals". Every non-terminal has a type of level at most 2. Moreover, there is a distinguished non-terminal S of level 0, called the start symbol.
- A finite set Σ of typed constants, called "terminal symbols". Every terminal symbol has a type of level at most 1.
- Maybe some binder symbol ℓ. If it is present, then for every variable x, a new constant ℓ_x of type $\iota \to \iota$ and a new constant v_x of type ι is present. We consider any occurrence of v_x in a term of the form $\ell_x t$ to be bound, and identify terms that are equal up to renaming of bound variables in the usual way. (However, we still formally consider the constants ℓ_x and v_x as additional terminal symbols.)
- A set of recursion equations of the form $N\overrightarrow{x} = e$, where N is a non-terminal, the $\overrightarrow{x}$ are pairwise distinct typed variables such that $N\overrightarrow{x}$ is a term of type ι and $e \in \mathcal{T}^{\iota}(\mathcal{N} \cup \Sigma, \{\overrightarrow{x}\})$.
  If a binder is present we require moreover, that every occurrence of a constant ℓ_x is in the context $\ell_x t$ and that every occurrence of v_x is bound in e.

Note that we do *not* insist that there is at most one equation for every non-terminal. In general a recursion scheme denotes a set of trees.

A level-1 or level-0 recursion scheme is a recursion scheme where all non-terminals have a type of level at most 1 or at most 0, respectively. A recursion scheme is said to be a "word recursion scheme", if all terminals have type $\iota \to \iota$; it is called "pure" if no binder is present.

Remark 11. Due to the native binding mechanism a non-pure recursion scheme currently is only a theoretical concept. However, in the only case where we need an effective construction on recursion schemes, that is for level-0 recursion schemes, we can always use the given names and needn't do any renaming.

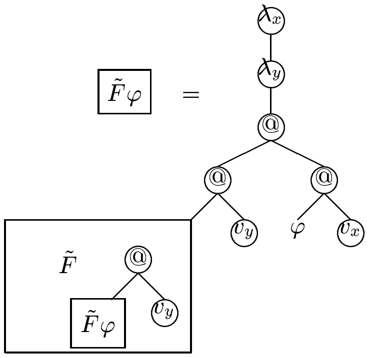
Example 12. Let $\Sigma = \{a, b, f, e\}$ a set of terminal symbols, each of which has type $\iota \to \iota$. For $\mathcal{N} = \{F, A, B, S, E\}$ a set of non-terminals with F of type $(\iota \to \iota) \to \iota \to \iota \to \iota$ and E, A, B, S of type ι. The following equations define a pure level-2 recursion scheme.

$$\begin{array}{lll} F\varphi xy = F(F\varphi y)y(\varphi x) & A = aE & S = FfAB \\ F\varphi xy = y & B = bE & E = eE \end{array}$$

Here x and y are variables of type ι and φ is a variable of type $\iota \to \iota$. It should be noted that this recursion scheme is neither safe nor deterministic.

Let $\tilde{\Sigma} = \Sigma \cup \{@\}$ with @ of type $\iota \to \iota \to \iota$ and λ a binder. Let $\tilde{F}$ be of type $\iota \to \iota$. Then a (non-pure) level-1 recursion scheme over $\tilde{\Sigma}$ with non-terminals $\tilde{\mathcal{N}} = \{\tilde{F}, \tilde{A}, \tilde{B}, \tilde{S}, \tilde{E}\}$ would be given by

$$\begin{aligned}\tilde{F}\varphi = \lambda_x(\lambda_y(@\ &(@(\tilde{F}(@(\tilde{F}z)v_y))v_y)\\ &(@\varphi v_x)))\end{aligned}$$



where φ is a variable of type ι. The other equations are

$$\tilde{F}\varphi = \lambda_x(\lambda_y y)$$

$$\tilde{A} = @a\tilde{E} \qquad \tilde{B} = @b\tilde{E}$$

$$\tilde{S} = @(@(\tilde{F}f)\tilde{A})\tilde{B} \qquad \tilde{E} = @e\tilde{E}$$

In fact, this is the reduced scheme (Definition 18) of the first scheme.

Proposition 13. *Let $t \in T^\tau(\mathcal{N}, \mathcal{X})$, with $\mathrm{lv}(\tau) \le 1$ and all variables in $\mathcal{X}$ of level at most 1 and all symbols in $\mathcal{N}$ of level at most 2. Then every occurrence of an $N \in \mathcal{N}$ is in a context $N\overrightarrow{t}$ such that $N\overrightarrow{t}$ has type of level at most 1.*

Corollary 14. *In every term e at the right hand side of a recursion equation of a level-2 recursion scheme it is the case that every non-terminal has all its level-1 arguments present.*

Remark 15. Corollary 14 shows that we may assume without loss of generality that in every non-terminal symbol of a level-2 recursion scheme the higher order arguments come first. In other words, at level-2 we can assume types to be homogeneous [7]. From now on we will use this assumption tacitly.

Unwinding a recursion scheme yields a (potentially non well-founded) tree, labelled with terminal symbols. Following ideas of Knapik *et al* [6] we will now define the lambda tree associated with a level-2 pure recursion scheme. It will be generated by a level 1 grammar, the "reduced recursion scheme".

Definition 16 (reduced type of level 2). For $\tau = \tau_1 \to \ldots \to \tau_n \to \iota \to \ldots \to \iota \to \iota$ with the τ_i types of level 1 we define inductively $\tau' = \tau_1' \to \ldots \to \tau_n' \to \iota$.

Definition 17 (reduced terms). For any term $e \in \mathcal{T}(\mathcal{N} \cup \Sigma, \{\overrightarrow{x}\})$ we define a reduced term e' inductively as follows.

$$\begin{aligned}(C\overrightarrow{t}\,\overrightarrow{s})' &= (C\overrightarrow{t'})@s_1'@\ldots@s_n'\\ (\varphi\overrightarrow{t})' &= \varphi@t_1'@\ldots@t_n'\\ x' &= v_x\end{aligned}$$

Here $C \in \mathcal{N} \cup \Sigma$ is a non-terminal or a letter with $\overrightarrow{t}$ terms of type of level 1 and $\overrightarrow{s}$ terms of level 0; $\varphi \in \{\overrightarrow{x}\}$ a variable of non-ground type with arguments $\overrightarrow{t}$ necessarily of type of level 0 and x a variable of ground type; @ is a terminal of type $\iota \to \iota \to \iota$, here written in infix.

Definition 18 (reduced recursion scheme). For a level-2 recursion scheme, where we assume without loss of generality that all non-terminals have their first-order arguments first, we define a level-1 recursion scheme, the *reduced recursion scheme*, defining a lambda tree in the following way.

Terminals are those of the original recursion scheme, however with reduced type, and a new symbol $@\colon \iota \to \iota \to \iota$. The new recursion scheme has a binder λ. Non-terminals are the same as in the original, however with the type reduced.

For any rule of the original recursion scheme of the form $N\overrightarrow{\varphi}\overrightarrow{x} \to e$ where $\overrightarrow{\varphi}$ are the level-1 arguments and $\overrightarrow{x}$ the level-0 arguments we add a rule $N\overrightarrow{\varphi} \to \overrightarrow{\lambda_x}.e'$ with e' being the reduced term of e.

Definition 19 (typed lambda trees). A *typed lambda tree* is a lambda tree with nodes labelled by simple types in such a way that

- Every variable is bound and if it is labelled with type σ, then the matching lambda has type $\sigma \to \tau$ for some τ; moreover, every abstraction node is labelled with a type of the form $\sigma \to \tau$ and its child is labelled with τ.
- If an application is labelled τ then, for some type σ the left child is labelled by $\sigma \to \tau$ and the right child by σ.
- Every terminal $f \in \Sigma$ is labelled by its type.

A lambda tree is *finitely typed* if it is a typed lambda tree and all types come from the same finite set.

Proposition 20. *The reduced recursion scheme of a pure level-*2 *word recursion scheme is finitely typed.*

Proposition 21. *If a lambda tree is finitely typed then the difference in the number of lambdas and applications seen on the matching path of an abstraction is bounded.*

Corollary 22. *In the reduced recursion scheme of a pure level-*2 *word recursion scheme the matching argument of an abstraction can be found by a finite state path walking automaton.*

4 Reductions and Denotations

In this section we make precise our intuitive notion of the tree denoted of recursion scheme, the (ω-word or) tree denoted by a lambda tree, and prove the needed properties. This essentially recasts results of Knapik *et al* [6].

Definition 23 (reduction relation). Let $\mathcal{S}$ be a recursion scheme. Its associated reduction relation $\to_{\mathcal{S}}$ on finite terms is inductively defined as follows.

- $N\overrightarrow{t} \to_{\mathcal{S}} e\left[\overrightarrow{t}/\overrightarrow{x}\right]$ if there is a recursion equation $N\overrightarrow{x} = e$ in $\mathcal{S}$.
- If $t \to_{\mathcal{S}} t'$ then $ts \to_{\mathcal{S}} t's$. If f is a terminal symbol and $t \to_{\mathcal{S}} t'$ then $f\overrightarrow{s}t \to_{\mathcal{S}} f\overrightarrow{s}t'$.

– In particular we have: If $t \to_{\mathcal{S}} t'$ then $\lambda_x t \to_{\mathcal{S}} \lambda_x t'$.

If in any reduction a variable is substituted in the scope of a binder then appropriate renaming is assumed to prevent the variable from getting bound at a new binder. In other words our reduction is capture avoiding in the usual sense. As we identify α-equal terms this is a well-defined notion.

Intuitively $t^{\perp}$ is t with all unfinished computations replaced by $\perp$. More precisely we have the following

Definition 24 (constructed part of a term). For t a term we define $t^{\perp}$ inductively as follows.

- $f^{\perp} = f$ for $f \in \Sigma$ a terminal and $N^{\perp} = \perp$ for $N \in \mathcal{N}$ a non-terminal.
- If $s^{\perp} = \perp$ then $(st)^{\perp} = \perp$. Otherwise $(st)^{\perp}=s^{\perp}t^{\perp}$.
- In particular we have $(\lambda_x t)^{\perp} = \lambda_x t^{\perp}$.

Moreover, we define a partial order $\sqsubseteq$ on terms inductively as follows.

- $\perp \sqsubseteq t$ and $f \sqsubseteq f$ for every constant f.
- If $s \sqsubseteq s'$ and $t \sqsubseteq t'$ then $st \sqsubseteq s't'$.
- In particular: If $t \sqsubseteq t'$ then $\lambda_x t \sqsubseteq \lambda_x t'$.

This order is obviously reflexive, transitive and directed complete.

Lemma 25. *If $t \to_{\mathcal{S}} t'$ then $t^{\perp} \sqsubseteq t'^{\perp}$.*

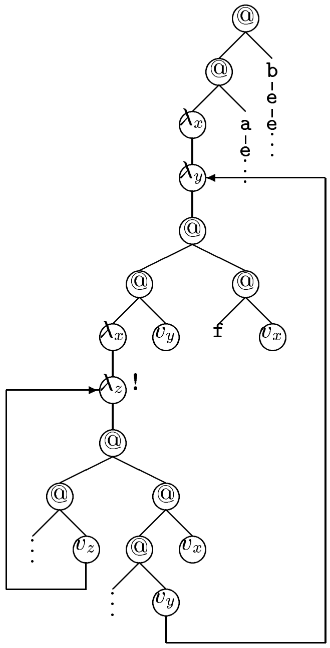
Definition 26 (terms over a signature). The set $\mathcal{T}^{\infty}(\Sigma)$ of not necessarily well founded terms over the signature Σ is coinductively defined by "If $t \in \mathcal{T}^{\infty}(\Sigma)$ then $t = f\overrightarrow{t}$ for some $t_1, \ldots, t_n \in \mathcal{T}^{\infty}(\Sigma)$ and $f \in \Sigma$ of type $\underbrace{\iota \to \ldots \to \iota}_{n} \to \iota$".

Definition 27 (language of a recursion scheme). Let $\mathcal{S}$ be a recursion scheme with start symbol S. Then $t \in \mathcal{T}^{\infty}(\Sigma)$ *belongs to the language of* $\mathcal{S}$, in symbols $t \in L(\mathcal{S})$, if t is finite and there are terms $S = t_0 \to_{\mathcal{S}} t_1 \to_{\mathcal{S}} \ldots \to_{\mathcal{S}} t_n = t$; or t is infinite and there are terms $S = t_0 \to_{\mathcal{S}} t_1 \to_{\mathcal{S}} \ldots$ where t is the supremum of the $t_i^{\perp}$.

It should be noted that $\mathcal{T}^{\infty}(\Sigma)$, and hence the language of a recursion scheme, contains only total objects. This, however, is not a restriction, as introducing a new terminal $f_{\mathcal{R}} \colon \iota \to \iota$ and transforming every rule $N\overrightarrow{x} = e$ to $N\overrightarrow{x} = f_{\mathcal{R}}e$ guarantees the defined trees to be total. Note moreover, that "removing the repetition constructors $f_{\mathcal{R}}$" is MSO definable.

Remark 28 (Status of binder). If a binder is present in a recursion scheme, then Definition 27 is only defined up to renaming of bound variables, as α-equal terms are identified. However, as all our notions on lambda trees are invariant under α-equality we can safely assume that some canonical α-variants of the generated lambda trees are chosen.

We note moreover that in the only case where we actually need this construction to be effective, that is in the case of a level-0 recursion scheme, only closed terms are replaced by other closed terms, so we can (and will) always choose the variable named in the underlying equation.



Example 29. The figure on the left shows a lambda tree of the language given by the second grammar in Example 12 with added arrows indicating some of the bindings; note in particular the renaming at the marked node. This example will be continued in Example 41.

Proposition 30 (level-0 trees). *Let $\mathcal{S}$ be a level-0 recursion scheme. Since in its reductions no capture can occur (since there are no substitutions), we may assume it to be pure (and consider λ_x and v_x as usual terminals). Then $L(\mathcal{S})$ is the language of a Büchi tree automaton.*

Definition 31 (reduction relation with beta). Let $\mathcal{S}'$ be a recursion scheme with binder λ and distinguished binary terminal @. We define a relation $\rightarrow^{\beta}_{\mathcal{S}'}$ inductively as follows.

- $N\overrightarrow{t} \rightarrow^{\beta}_{\mathcal{S}'} e\left[\overrightarrow{t}/\overrightarrow{x}\right]$ if there is a recursion equation $N\overrightarrow{x} = e$ in $\mathcal{S}$.
- $(\lambda_x t)@s \rightarrow^{\beta}_{\mathcal{S}'} t\,[s/v_x]$. We recall that our notion of substitution is understood to be capture free.
- If $t \rightarrow^{\beta}_{\mathcal{S}'} t'$ then $t@s \rightarrow^{\beta}_{\mathcal{S}'} t'@s$. Moreover, if t is of the form $f@r_1@\ldots@r_n$ and $s \rightarrow s'$ then $t@s \rightarrow^{\beta}_{\mathcal{S}'} t@s'$.

We also define $t^{\perp\beta}$ as t with all unfinished computations replaced by $\perp$. This time, however, we consider @ as denoting application and the $f \in \Sigma$ as belonging to a reduced recursion scheme. That is we want $((\lambda_x \ldots)@a)^{\perp\beta} = \perp$ and $(f@a)^{\perp\beta} = fa$.

Definition 32 (constructed part of a recursion scheme with beta). For t a term we define $t^{\perp\beta}$ inductively using the "vector notation" characterisation of terms, provided by Proposition 9.

- $f^{\perp\beta} = f$ for $f \in \Sigma$ and $\left(N\overrightarrow{t}\right)^{\perp\beta} = (\lambda_x t)^{\perp\beta} = \perp$.
- If $t^{\perp\beta} = f\overrightarrow{r}$ for some $f \in \Sigma$ then $(t@s)^{\perp\beta} = t^{\perp\beta}s^{\perp\beta}$; otherwise $(t@s)^{\perp\beta} = \perp$.

Lemma 33. *If $t \rightarrow^{\beta}_{\mathcal{S}'} t'$ then $t^{\perp\beta} \sqsubseteq t'^{\perp\beta}$.*

Definition 34 (The beta-language of a recursion scheme). Let $\mathcal{S}'$ be a recursion scheme with start symbol S, binder λ and distinguished non-terminal @. Then $t \in \mathcal{T}^\infty(\Sigma)$ *belongs to the β-language of* $\mathcal{S}'$, in symbols $t \in L^\beta(\mathcal{S}')$, if t is finite and there are terms $S = t_0 \rightarrow^\beta_{\mathcal{S}'} t_1 \rightarrow^\beta_{\mathcal{S}'} \ldots \rightarrow^\beta_{\mathcal{S}'} t_n$ with $t_n^{\perp\beta} = t$, or t is infinite and there are terms $S = t_0 \rightarrow^\beta_{\mathcal{S}'} t_1 \rightarrow^\beta_{\mathcal{S}'} \ldots$ and t is the supremum of the $t_i^{\perp\beta}$.

Lemma 35. *Let $\mathcal{S}$ be a recursion scheme and $\mathcal{S}'$ the reduced scheme. Then $w \in L(\mathcal{S}) \Leftrightarrow w \in L^\beta(\mathcal{S}')$.*

Proof. First we note that $N\overrightarrow{s}\overrightarrow{t} \rightarrow^\beta_{\mathcal{S}'} (\overrightarrow{\lambda_x}.e\,[\overrightarrow{s}/\overrightarrow{\varphi}])@t_0@\ldots@t_n \rightarrow^\beta_{\mathcal{S}'} \ldots \rightarrow^\beta_{\mathcal{S}'} e\left[\overrightarrow{s}\overrightarrow{t}/\overrightarrow{\varphi}\overrightarrow{x}\right]$. So a reduction sequence for $\mathcal{S}$ approximating some term $w \in L(\mathcal{S})$ can be transformed into a reduction sequence for $\mathcal{S}'$ approximating the same $w \in L(\mathcal{S}')$.

Moreover, we note that if in $\mathcal{S}'$ we have that $t \rightarrow^\beta_{\mathcal{S}'} t'$ by unfolding of a non-terminal, that is, by replacing somewhere in t the expression $N\overrightarrow{s}$ by $\overrightarrow{\lambda_x}.e\,[\overrightarrow{s}/\overrightarrow{\varphi}]$ then this happens in a context to produce $r = (\overrightarrow{\lambda_x}.e\,[\overrightarrow{s}/\overrightarrow{\varphi}])@t_1@\ldots@t_n$. Moreover $r^{\perp\beta} = \perp$ and every $\rightarrow^\beta_{\mathcal{S}'}$-reduction sequence starting with r has to first reduce the obvious beta redexes yielding $e\left[\overrightarrow{s},\overrightarrow{t}/\overrightarrow{\varphi},\overrightarrow{x}\right]$ with all reducts r' before obtaining $r'^{\perp\beta} \neq \perp$.

Lemma 36. *Let $\mathcal{S}'$ be the reduced recursion scheme of a pure level-2 recursion scheme $\mathcal{S}$. Then the word or tree met on the canonical traversal of the lambda-tree denoted by $\mathcal{S}'$ and the word denoted by $\mathcal{S}'$ coincide.*

Corollary 37. *For a pure recursion scheme, its language can also be described as the words met on the canonical traversals of the trees obtained by the reduced recursion scheme.*

5 First order functions

Recall that our main idea is to simulate an automaton walking along the word or tree met on the canonical traversal. By Corollary 37 this suffices to test for a given ω-regular property. However, we currently have two unsolved problems.

- The definition of the tree denoted by a first-order recursion scheme involves binders and substitution is assumed to be performed in a capture avoiding manner. This no longer results in local conditions on testing whether a tree is generated by a given recursion scheme.
- There is no obvious way to characterise level-1 trees by automata. So the idea of intersecting an automaton testing whether the tree belongs to the language of the recursion scheme and an automaton simulating the Büchi-automaton does not work.

Fortunately, both problems simultaneously disappear if we perform the same trick again, and denote parts of the lambda tree by second-order abstractions.

Definition 38 (second order lambda trees). A *second-order lambda tree* is a tree built from the following constructors.

- Letters f, g, h, …, "application" $\underline{@}$, "abstraction" λ_x and "variables" v_x
- "function abstraction $\underline{\lambda}\varphi$" and "function variables φ" and "function application $\underline{@}$".

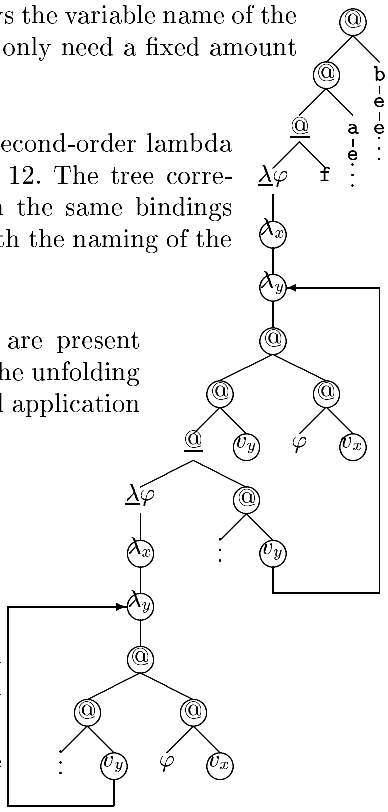
Definition 39 (typed second order lambda trees). A second-order lambda tree is called *typed* if it arises from a level-2 recursion scheme by replacing every rule $N\overrightarrow{\varphi}\overrightarrow{x} \to e$ by

$$N \to \overrightarrow{\underline{\lambda}\varphi}\,\overrightarrow{\lambda_x}\,e''$$

where the $\overrightarrow{\varphi}$ are the first-order arguments of N and the $\overrightarrow{x}$ are the ground type arguments of N. Here e'' is the term e reduced twice in the sense of Definition 17; the first time with respect to λ and @ and the second time with respect to $\underline{\lambda}$ and $\underline{@}$.

Remark 40. Note that in a level-0 recursion scheme no substitution is performed and hence no variable renaming is necessary. Therefore, in accordance with Remark 28, we assume that always the variable name of the underlying equation is chosen. In particular we only need a fixed amount of variables.

Example 41. The figure on the right shows a second-order lambda tree associated with the grammar in Example 12. The tree corresponds to the one shown in Example 29, with the same bindings indicated. Note that no care had to be taken with the naming of the variables.



Remark 42. Since all higher order arguments are present and the function abstractions $\underline{\lambda}\varphi$ come first in the unfolding of every nonterminal, first-order abstraction and application come always in the pattern

$$(\underline{\lambda}\overrightarrow{\varphi}.t)\underline{@}t_1\underline{@}\ldots\underline{@}t_n$$

which we will in the following abbreviate by

$$\underline{\mathrm{subst}}(t, \overrightarrow{\varphi}, \overrightarrow{t})$$

It should be noted that –as only finitely many arities are possible– this introduction of new $\underline{\mathrm{subst}}(\cdot, \overrightarrow{\varphi}, \cdot)$ constructor symbols does not destroy the automata-theoretic description of the tree language.

Proposition 43. *Consider a typed lambda tree for words, that is, with every letter $f \in \Sigma$ of type $\iota \to \iota$. The canonical traversal continues in one of the following ways when entering a subterm of type $\underbrace{\iota \to \ldots \to \iota}_{k} \to \iota$.*

– *continues forever within this subterm*
– *passes finitely many letters and eventually hits a variable denoting one of the k function arguments*
– *it passes finitely letters and eventually hits a free variable of the subterm*

The crucial point about Proposition 43 is that it shows that only a *fixed amount of information* is needed to describe a first-order variable φ. This will allow us to split the traversal into two parts in the style of a logical "cut", and non-deterministically guess the splittings, carrying our guesses in the state of the automaton.

6 Word Languages

Theorem 44. *For any ω-regular property of words, fixed set of types and variables, there is a Büchi tree automaton that accepts precisely those second order lambda trees over the given set of variables and typed by the given types, where the denoted word satisfies the regular property. (We do not care what the automaton does on input trees that are not appropriately typed second-order lambda-trees.)*

Proof. Let the ω-regular property be given as a nondeterministic Büchi word automaton. First assume that there are no first-order variables. Then what we have to do is to assign each of the visited letters the state it has in a successful run (which we guess) and check local correctness and acceptance condition.

To make the transitions completely local, we use the fact that every node is visited at most three times so that we can guess for each node up to three annotations of the form "a automaton coming from direction ... in state ... is continuing its path here", "... is searching upwards for variable x" or "... is looking for the k'th argument". Since the number of arguments is bounded by the type for every such statement only a fixed amount of information is needed and the correctness of the guessed traversal can be checked locally.

Concerning the acceptance condition: given that each node of the lambda tree is visited only finitely many times, it must be the case that we visit infinitely many nodes in order to traverse an infinite word. As the automaton moves locally, at every node where an automaton enters, but doesn't come back there must be a child where an automaton enters but doesn't come back. Following these nodes traces a path that is visited infinitely often by the simulated automaton. Acceptance checking results in only signalling acceptance on the distinguished path (which we can guess) only if the automaton visits it having visited an accepting state since the last visit of the distinguished path.

Now assume that first-order variables are present. If we then consider walking just down the $\underline{\mathrm{subst}}(t, \overrightarrow{\varphi}, \overrightarrow{t})$ nodes as if they were not present, we might at some point hit a first-order variable φ, that stands for a tree. The path might either continue forever in the tree denoted by φ or come back after finitely many steps asking to continue in one of its k arguments.

We note that there is still a path that is visited infinitely often: either the main branch of a $\underline{\text{subst}}(\cdot, \overrightarrow{\varphi}, \cdot)$ node or the subtree the variable stands for where the automaton enters, but never returns.

What remains to be shown is that local correctness can be tested. To do so, we keep for every of the (finitely many!) first-order variables a table with our guess of the behaviour of the variable in the current context. Such an entry is one of the following.

- "The variable will not be needed."
- "The successful run will enter in state q and will remain there forever."
- "When entering this variable the path will come back asking for argument k and the transition table for the word read in between is δ."

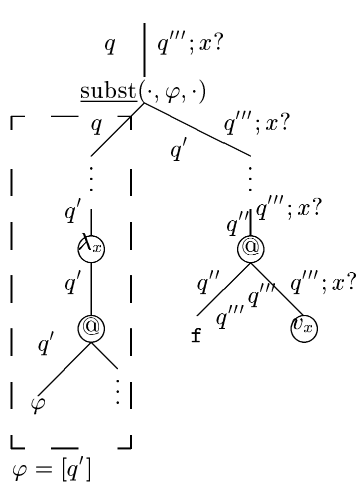
Here a "transition table" is understood as a table saying for every pair (p, q) of states of the simulated automaton, whether, when entering with state p we can leave with state q and whether we can do so with a visit of an accepting state in between.

Obviously only a fixed amount of information is needed to store this table. At every $\underline{\text{subst}}(t, \overrightarrow{\varphi}, \overrightarrow{t})$-"node" we update the table for the variables $\overrightarrow{\varphi}$ and our offspring to subtree t_i verifies our guess for φ_i.

- For the guess "The variable will not be needed" we need to verify nothing. So we just accept this subtree.
- For the guess "The successful run will enter here with state q" we have to simulate a successful run starting in state q.

  Note that we will continue producing the successful run even if this involves going upwards beyond the $\underline{\text{subst}}(\cdot, \overrightarrow{\varphi}, \cdot)$ where we entered the side branch. *This is precisely the device that allows us to work without a safety condition.*

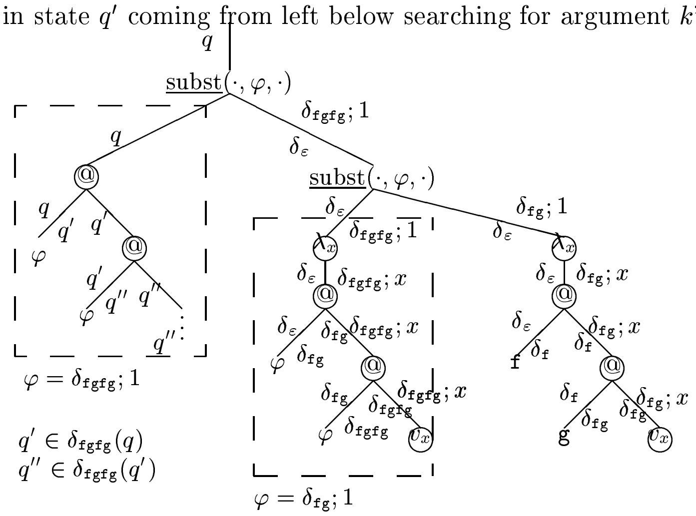
  The picture on the right shows an example of such a run in a situation where blind substitution would produce variable capture. It should be noted that above the $\underline{\text{subst}}(\cdot, \varphi, \cdot)$ node all the annotations are the same, as if the automaton looking for variable x came directly from the main branch, correctly jumping over the λ_x there.

- For the guess "will come back to argument k with transition table δ", we have to produce a symbolic run, i.e., a run with transition tables rather than just states. But as transition tables can be updated locally when reading a letter we can produce such a symbolic run in the same way, as we would produce a usual run.

All this can be checked locally. Here we note that the guessing and verifying mechanism can be used for first-order variables, even if we are in a branch that currently verifies a guess. The reason is that our guesses are absolute ones and we needn't care what we use them for. The figure after this proof shows an example of such a guessed and verified run.

When our path now hits a second order variable φ, we do the following, depending on our guess of how φ behaves. If we have guessed that φ will not be needed, we fail. If we have guessed that φ will be entered in state q then we accept if and only if the current state is q. If we have guessed that the path will come back asking for argument k, we choose a state q' in accordance with the guessed transition table and make sure the node above has annotation "an automaton in state q' coming from left below searching for argument k".



Corollary 45. *Given a level-2 word recursion scheme and an MSO property, it is decidable whether some word can be generated with the given recursion scheme that has the given property.*

7 Tree Languages

In this section we show how the proof of Theorem 44 can be extended to tree languages. To do so, we will use a result of Vardi [12] showing that the emptiness problem for *two way alternating parity tree automata* is decidable.

This has the advantage that we can follow the canonical paths directly (even upwards) and we can use our own alternating power to follow the alternations of the simulated automata and we use our own parity conditions to check the parity conditions of the simulated automaton. The only non-trivial point is still the case when we meet a first-order variable. Here we branch and do the same "guess and verify" as in the word case; this time with guesses of the form

> "There is a successful run in the tree denoted by φ, starting from state q with automata entering the arguments of this function with at most the given states and the given parities visited in between."

Theorem 46. *For any alternating parity tree automaton there is a two-way alternating parity tree automaton that accepts precisely those typed second order lambda-trees denoting trees that have an accepting run of the given automaton.*

Proof. Our aim is to simulate an accepting run of an alternating *one-way* parity tree automaton on a tree denoted by the recursion scheme.

Since we now can also walk backwards we follow the path directly and we use our own alternating power to do the alternating of the simulated automaton. The only thing that remains to show is what we do, if we hit a first-order variable.

In short, we guess what the run through that variable would look like and then, on the one hand, send an automaton upwards to verify the guess and on the other hand simulate those paths of the tree that come up from the variable again. Of course we must make sure that the guess can be stored in the state of the automaton walking upwards and that there is only a bounded number of possible automata coming out of the variable and continue their run here (so that we can use our alternating power to branch off in such a strong way). This is achieved by the following observations.

- Automata coming out in the same state, with the same set of parities visited, and asking for the same argument can be merged into a single automaton, as automata entering the same tree with the same condition either all have an accepting run, or none.
- The question whether the automata on the paths that never leave the (tree denoted by) the first-order variable have an accepting run can be checked by the automaton walking upwards towards the variable.

Hence our guess "There is a successful run with automata entering the arguments of this function with at most the given states" can be described by a subset of $Q \times \{1, \ldots, k\} \times \mathfrak{P}(\Omega)$ where Q is the set of states of the simulated automaton, k the arity of the variable (which is bounded by the maximal number of arrows in any non-terminal of the original grammar) and $\mathfrak{P}(\Omega)$ the power set of the set of parities of the original automaton. So our guess comes from a fixed finite set.

The automaton walking upwards verifies its guess in the following way. It enters in the designated start state and simulates a run in the usual way, except that it remembers in its states the following information.

- The fact that we are in verifying a guess and its minimal parity visited since entering here.
- The set of allowed state/argument/acceptance triples for leaving the subtree "upwards"

This information is only used when walking upwards and hitting the $\underline{\mathrm{subst}}(\cdot, \overrightarrow{\varphi}, \cdot)$-node from somewhere different than the main-branch, i.e., when leaving the subtree denoted by the variable. Then we check, whether we leave in one of the allowed states. Of course when hitting the $\underline{\mathrm{subst}}(\cdot, \overrightarrow{\varphi}, \cdot)$ node while searching for a variable, we continue our simulation (just forgetting that we were in verifying mode); we can do this, as this path will never ask for an argument of φ_i. *Again, this is the device, that allows us to work without a safety condition.*

Corollary 47. *The MSO theory of tree languages given by level-2 recursion schemes (that need not be safe or deterministic) is decidable.*

Proof. For every MSO formula there is an alternating parity tree automaton accepting those trees that satisfy it [11, 10]. Since level-0 recursion schemes are given by tree automata, we can intersect the language of the automaton obtained by Theorem 46 with the language of the corresponding level-0 recursion scheme. The decidability of the non-emptiness of two-way alternating parity tree automata was shown by Vardi [12].

References

1. Klaus Aehlig, Jolie G de Miranda, and C H Luke Ong. Safety is not a restriction at level two for string languages. In *Foundations of Software Science and Computation Structures (FOSSACS '05)*, April 2005. To appear.
2. A. Asperti, V. Danos, C. Laneve, and L. Regnier. Paths in the lambda-calculus. In *Proceedings of the Ninth Annual IEEE Symposium an Logic in Computer Science (LICS '94)*, pages 426–436, July 1994.
3. D. Caucal. On infinite transition graphs having a decidable monadic theory. In F. M. auf der Heide and B. Monien, editors, *Proceedings of the 23th International Colloquium on Automata, Languages and Programming (ICALP '96)*, volume 1099 of *Lecture Notes in Computer Science*, pages 194–205. Springer Verlag, 1996.
4. B. Courcelle. The monadic second-order logic of graphs IX: Machines and their behaviours. *Theoretical Comput. Sci.*, 151(1):125–162, 1995.
5. J. R. Kennaway, J. W. Klop, and F. J. d. Vries. Infinitary lambda calculus. *Theoretical Comput. Sci.*, 175(1):93–125, Mar. 1997.
6. T. Knapik, D. Niwiński, and P. Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In S. Abramsky, editor, *Proceedings of the 5th International Conference on Typed Lambda Caculi and Applications (TLCA '01)*, volume 2044 of *Lecture Notes in Computer Science*, pages 253–267. Springer Verlag, 2001.
7. T. Knapik, D. Niwiński, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielson, editor, *Proceedings of the 5th International Conference Foundations of Software Science and Computation Structures (FOSSACS '02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222, Apr. 2002.
8. T. Knapik, D. Niwiński, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars, panic automata, and decidability. Manuscript, Oct. 2004.
9. O. Kupferman and M. Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In E. A. Emerson and A. P. Sistla, editors, *12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 36–52. Springer Verlag, 2000.
10. D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Comput. Sci.*, 54:267–276, 1987.
11. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, July 1969.
12. M. Y. Vardi. Reasoning about the past with two-way automata. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP 98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer Verlag, 1998.
13. I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, Jan. 2001.