Complexity of Model Checking Recursion Schemes for Fragments of the Modal Mu-Calculus

Naoki Kobayashi[1] and C.-H. Luke Ong[2]

[1] Tohoku University
[2] University of Oxford

Abstract. Ong has shown that the modal mu-calculus model checking problem (equivalently, the alternating parity tree automaton (APT) acceptance problem) of possibly-infinite ranked trees generated by order-n recursion schemes is n-EXPTIME complete. We consider two subclasses of APT and investigate the complexity of the respective acceptance problems. The main results are that, for APT with a single priority, the problem is still n-EXPTIME complete; whereas, for APT with a disjunctive transition function, the problem is $(n-1)$-EXPTIME complete. This study was motivated by Kobayashi's recent work showing that the resource usage verification for functional programs can be reduced to the model checking of recursion schemes. As an application, we show that the resource usage verification problem is $(n-1)$-EXPTIME complete.

1 Introduction

The model checking problem for higher-order recursion schemes has been a topic of active research in recent years (for motivation as to why the problem is interesting, see e.g. the introduction of Ong's paper [1]). This paper studies the complexity of the problem with respect to certain fragments of the modal μ-calculus. A higher-order recursion scheme (recursion scheme, for short) is a kind of (deterministic) grammar for generating a possibly-infinite ranked tree. The model checking problem for recursion schemes is to decide, given an order-n recursion scheme $\mathcal{G}$ and a specification ψ for infinite trees, whether the tree generated by $\mathcal{G}$ satisfies ψ. Ong [1] has shown that if ψ is a modal μ-calculus formula (or equivalently, an alternating parity tree automaton), then the model checking problem is n-EXPTIME complete.

Following Ong's work, Kobayashi [2] has recently applied the decidability result to the model checking of higher-order functional programs (precisely, programs of the simply-typed λ-calculus with recursion and resource creation/access primitives). He considered the *resource usage verification problem* [3]—the problem of whether programs access dynamically created resources in a valid manner (e.g. whether every opened file will eventually be closed, and thereafter never read from or written to before it is reopened). He showed that the resource usage verification problem reduces to a model checking problem for recursion schemes

S. Albers et al. (Eds.): ICALP 2009, Part II, LNCS 5556, pp. 223–234, 2009.

by giving a transformation that, given a functional program, constructs a recursion scheme that generates all possible resource access sequences of the program. From Ong's result, it follows that the resource usage verification problem is in n-EXPTIME (where, roughly, n is the highest order of types in the program). This result also implies that various other verification problems, including (the precise verification of) reachability ("Given a closed program, does it reach the fail command?") and flow analysis ("Does a sub-term e evaluate to a value generated at program point l?"), are also in n-EXPTIME, as they can be easily recast as resource usage verification problems.

It was however unknown whether n-EXPTIME is the tightest upper-bound of the resource usage verification problem. Although the model checking of recursion schemes is n-EXPTIME-hard for the full modal μ-calculus, only a certain fragment of the modal μ-calculus is used in Kobayashi's approach to the resource usage verification problem. First, specifications are restricted to safety properties, which can be described by Büchi tree automata with a trivial acceptance condition (the class called "trivial automata" by Aehlig [4]). Secondly, specifications are also restricted to linear-time properties—the branching structure of trees is ignored, and only the path languages of trees are of interest. Thus, one may reasonably hope that there is a more tractable model checking algorithm than the n-EXPTIME algorithm.

The goal of this paper is, therefore, to study the complexity of the model checking of recursion schemes for various fragments of the modal μ-calculus (or, alternating parity tree automata) and to apply the result to obtain tighter bounds of the complexity of the resource usage verification problem.

The main results of this paper are as follows:

– The problem of whether a given Büchi automaton with a trivial acceptance condition (or, equivalently, alternating parity tree automaton with a single priority 0) accepts the tree generated by an order-n recursion scheme is still n-EXPTIME-hard. This follows from the n-EXPTIME-completeness of the word acceptance problem for higher-order alternating pushdown automata[1] [5].

– We introduce a new subclass of alternating parity tree automata (APT) called *disjunctive APT*, and show that its acceptance problem for trees generated by order-n recursion schemes is $(n-1)$-EXPTIME complete. From this general result, it follows that both the linear-time properties (including reachability, which is actually $(n-1)$-EXPTIME complete) and finiteness of the tree generated by a recursion scheme are $(n-1)$-EXPTIME.

– As an application, we show that the resource usage verification problem [2] is also $(n-1)$-EXPTIME-complete, where n is the highest order of types used in the source program (written in an appropriate language [2]).

Related Work. For the class of Büchi automata with a trivial acceptance condition, Kobayashi [2] showed that the complexity is linear in the size of recursion schemes, if the sizes of types and automata are bounded above by a

[1] Engelfriet's proof [5] is for a somewhat different—but equivalent—machine which is called *iterated pushdown automaton*.

constant. For the full modal μ-calculus, Kobayashi and Ong [6] have shown that the complexity is polynomial-time in the size of the recursion scheme, assuming that the size of types and the formula are bounded above by a constant.

2 Preliminaries

We assume the standard notions of (ranked/unranked) infinite trees [1].

Higher-Order Recursion Schemes. The set of *types* is defined by: $\kappa ::= \mathsf{o} \mid \kappa_1 \to \kappa_2$, where o describes trees. The *order* of κ, written $order(\kappa)$, is defined by: $order(\mathsf{o}) := 0$ and $order(\kappa_1 \to \kappa_2) := max(order(\kappa_1) + 1, order(\kappa_2))$. A (deterministic) higher-order recursion scheme (recursion scheme, for short) is a quadruple $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$, where (i) Σ is a *ranked alphabet* i.e. a map from a finite set of symbols called *terminals* to types of order 0 or 1. (ii) $\mathcal{N}$ is a map from a finite set of symbols called *non-terminals* to types. (iii) $\mathcal{R}$ is a set of rewrite rules $F\ \widetilde{x} \to t$. Here, $\widetilde{x}$ abbreviates a sequence of variables, and t is an applicative term constructed from non-terminals, terminals, and variables. (iv) S is a *start symbol*. We require that $\mathcal{N}(S) = \mathsf{o}$. The set of (typed) terms is defined in the standard manner: A symbol (i.e., a terminal, non-terminal, or variable) of type κ is a term of type κ. If terms t_1 and t_2 have types $\kappa_1 \to \kappa_2$ and κ_1 respectively, then $t_1\ t_2$ is a term of type κ_2. For each rule $F\ \widetilde{x} \to t$, $F\ \widetilde{x}$ and t must be terms of type o. There must be exactly one rewrite rule for each non-terminal. The *order* of a recursion scheme is the highest order of its non-terminals.

A rewrite relation on terms is defined inductively by: (i) If $F\,\widetilde{x} \to t \in \mathcal{R}$, then $F\,\widetilde{s} \longrightarrow_{\mathcal{G}} [\widetilde{s}/\widetilde{x}]t$. (ii) If $t \longrightarrow_{\mathcal{G}} t'$, then $t\,s \longrightarrow_{\mathcal{G}} t'\,s$ and $s\,t \longrightarrow_{\mathcal{G}} s\,t'$. The *value tree* of a recursion scheme $\mathcal{G}$, written $[\![\mathcal{G}]\!]$, is the (possibly infinite) tree obtained by infinite rewriting of the start symbol S: See [1] for a precise definition.

Alternating parity tree automata. Given a finite set X, the set $\mathsf{B}^+(X)$ of *positive Boolean formulas* over X is defined as follows:

$$\mathsf{B}^+(X) \ni \theta\ ::=\ \mathsf{t} \mid \mathsf{f} \mid x \mid \theta \wedge \theta \mid \theta \vee \theta$$

where x ranges over X. We say that a subset Y of X *satisfies* θ just if assigning true to elements in Y and false to elements in $X \setminus Y$ makes θ true.

An *alternating parity tree automaton* (or APT for short) over Σ-labelled trees is a tuple $\mathcal{A} = (\Sigma, Q, \delta, q_I, \Omega)$ where (i) Σ is a ranked alphabet; let m be the largest arity of the terminal symbols; (ii) Q is a finite set of states, and $q_I \in Q$ is the initial state; (iii) $\delta : Q \times \Sigma \longrightarrow \mathsf{B}^+(\{1, \ldots, m\} \times Q)$ is the transition function where, for each $f \in \Sigma$ and $q \in Q$, we have $\delta(q, f) \in \mathsf{B}^+(\{1, \ldots, arity(f)\} \times Q)$; and (iv) $\Omega : Q \longrightarrow \{0, \cdots, M-1\}$ is the priority function.

A *run-tree* of an APT $\mathcal{A}$ over a Σ-labelled ranked tree T is a $(dom(T) \times Q)$-labelled unranked tree r satisfying: (i) $\epsilon \in dom(r)$ and $r(\epsilon) = (\epsilon, q_I)$; and (ii) for every $\beta \in dom(r)$ with $r(\beta) = (\alpha, q)$, there is a set S that satisfies $\delta(q, T(\alpha))$; and for each $(i, q') \in S$, there is some j such that $\beta\, j \in dom(r)$ and $r(\beta\, j) = (\alpha\, i, q')$.

Let $\pi = \pi_1\, \pi_2 \cdots$ be an infinite path in r; for each $i \geq 0$, let the state label of the node $\pi_1 \cdots \pi_i$ be q_{n_i} where q_{n_0}, the state label of ϵ, is q_I. We say that

π satisfies the *parity* condition just if the largest priority that occurs infinitely often in $\Omega(q_{n_0})\,\Omega(q_{n_1})\,\Omega(q_{n_2})\cdots$ is even. A run-tree r is *accepting* if every infinite path in it satisfies the parity condition. An APT $\mathcal{A}$ accepts a (possibly infinite) ranked tree T if there is an accepting run-tree of $\mathcal{A}$ over T.

Ong [1] has shown that there is a procedure that, given a recursion scheme $\mathcal{G}$ and an APT $\mathcal{A}$, decides whether $\mathcal{A}$ accepts the value tree of $\mathcal{G}$.

Theorem 1 (Ong). *Let $\mathcal{G}$ be a recursion scheme of order n, and $\mathcal{A}$ be an APT. The problem of deciding whether $\mathcal{A}$ accepts $[\![\mathcal{G}]\!]$ is n-EXPTIME-complete.*

3 Trivial APT and the Complexity of Model Checking

APT with a trivial acceptance condition, or *trivial APT* (for short), is an APT that has exactly one priority which is even. Note that trivial APT are equivalent to Aehlig's "trivial automata" [4] (for defining languages of ranked trees).

The first result of this paper is a logical characterization of the class of ranked trees accepted by trivial APT. Call $\mathcal{S}$ the following "safety fragment" of the modal mu-calculus:

$$\phi, \psi \ ::= \ P_f \mid Z \mid \phi \wedge \psi \mid \phi \vee \psi \mid \langle i \rangle \phi \mid \nu Z.\phi$$

where f ranges over symbols in Σ, and i ranges over $\{1, \cdots, \mathit{arity}(\Sigma)\}$.

Proposition 1 (Equi-Expressivity). *The logic $\mathcal{S}$ and trivial APT are equivalent for defining possibly-infinite ranked trees. I.e. for every closed $\mathcal{S}$-formula, there is a trivial APT that defines the same tree language, and vice versa.*

3.1 n-EXPTIME Completeness

We show that the model checking problem for recursion schemes is n-EXPTIME complete for trivial APT. The upper-bound of n-EXPTIME follows immediately from Ong's result [1]. To show the lower-bound, we reduce the decision problem of $w \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$, where w is a word and $\mathcal{A}$ is an order-n alternating PDA, to the model checking problem for recursion schemes. n-EXPTIME hardness follows from the reduction, since the problem of $w \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$ is n-EXPTIME hard [5].

Definition 1. An *order-n alternating PDA* (order-n APDA, for short) for finite words is a 7-tuple:

$$\mathcal{A} \ = \ \langle P,\ \lambda,\ p_0 \in P,\ \Gamma,\ \Sigma,\ \Delta \subseteq P \times \Gamma \times (\Sigma \cup \{\epsilon\}) \times P \times Op_n,\ F \subseteq P \rangle$$

where P is a set of states, $\lambda \in P \to \{\mathtt{A}, \mathtt{E}\}$, p_0 is the initial state, Γ is the set of stack symbols, Σ is an input alphabet, F is the set of final states, and Δ is a transition relation that satisfies: for every p, γ, if $(p, \gamma, \epsilon, p', \theta) \in \Delta$ for some p', θ, then $(p, \gamma, a, p', \theta) \notin \Delta$ for every $a \in \Sigma, p'$ and θ. A *configuration* of an order-n APDA is of the form (p, s) where s is an order-n stack: an order-1 stack

is an ordinary stack, and an order-$(k+1)$ stack is a stack of order-k stacks. The (induced) transition relation on configurations is defined by the rule:

$$\text{if } (p, top_1(s), \alpha, p', \theta) \in \Delta, \text{ then } (p, s) \longrightarrow_\alpha (p', \theta(s)).$$

Here, $\theta \in Op_n$ is an order-n stack operation and $top_1(s)$ is the stack top of s. The definition of the stack operations Op_n is omitted since it is not important for understanding the encodings below; interested readers may wish to consult, for example, the paper [8] by Knapik et al.

Let w be a word over Σ. We write w_i $(0 \leq i < |w|)$ for the i-th element of w. A *run tree* of an order-n APDA over a word w is a *finite*, unranked tree such that (i) The root is labelled by $(p_0, \bot_n, 0)$, where $\bot_n$ is the initial stack. (ii) If a node is labelled by (p, s, i) and $\lambda(p) = \mathtt{A}$, then either $p \in F$ and $i = |w|$, or the set of labels of the child nodes is exactly $\{(p', \theta(s), i+1) \mid (p, top_1(s), w_i, p', \theta) \in \Delta \wedge i < |w|\} \cup \{(p', \theta(s), i) \mid (p, top_1(s), \epsilon, p', \theta) \in \Delta\}$. (Thus, if the set is empty, the node has no child.) (iii) If a node is labelled by (p, s, i) and $\lambda(p) = \mathtt{E}$, then either $p \in F$ and $i = |w|$, or there exists exactly one child node which is labelled by an element of the set: $\{(p', \theta(s), i+1) \mid (p, top_1(s), w_i, p', \theta) \in \Delta \wedge i < |w|\} \cup \{(p', \theta(s), i) \mid (p, top_1(s), \epsilon, p', \theta) \in \Delta\}$. An order-$n$ APDA $\mathcal{A}$ *accepts* w if there exists a run tree of $\mathcal{A}$ over w.

Engelfriet [5] has shown that the word acceptance problem for order-n APDA is n-EXPTIME complete.

Theorem 2 (Engelfriet). *Let $\mathcal{A}$ be an order-n APDA and w a finite word over Σ. The problem of $w \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$ is n-EXPTIME complete.*

To reduce the word acceptance problem of order-n APDA to the model checking problem for recursion schemes, we use the equivalence [8] between order-n *safe* recursion schemes and order-n PDA as (deterministic) devices for generating trees.

Definition 2. An *order-n tree-generating (deterministic) PDA* is a 5-tuple $\langle \Sigma, \Gamma, Q, \delta, q_0 \rangle$ where Σ is a ranked alphabet, Γ is a finite stack alphabet, Q is a finite state-set, $\delta : Q \times \Gamma \longrightarrow (Q \times Op_n + \{(f; q_1, \cdots, q_{arity(f)}) : f \in \Sigma, q_i \in Q\})$ is the transition function, and $q_0 \in Q$ is the initial state. A *generalized configuration* is either a configuration (which has the shape (q, s) where s is an order-n stack over Γ) or a triple of the form $(f; q_1, \cdots, q_{arity(f)}; s)$. We define $\overset{\ell}{>}$, a labelled transition relation over generalized configurations, as follows:

- $(q, s) \overset{(q', \theta)}{>} (q', \theta(s))$ if $\delta(q, top_1(s)) = (q', \theta)$
- $(q, s) \overset{f\,\widetilde{q}}{>} (f; \widetilde{q}; s)$ if $\delta(q, top_1\, s) = (f; \widetilde{q})$
- $(f; \widetilde{q}; s) \overset{(f,i)}{>} (q_i, s)$ for each $1 \leq i \leq arity(f)$.

A *computation path* of an order-n PDA $\mathcal{A}$ is a finite or infinite transition sequence $\rho = c_0 \overset{\ell_0}{>} c_1 \overset{\ell_1}{>} c_2 \overset{\ell_2}{>} \cdots$ where each c_i is a generalized configuration, and

$c_0 = (q_0, \bot_n)$ is the initial configuration. The *Σ-projection* of ρ is the subsequence $\ell_{r_1}\ell_{r_2}\ell_{r_3}\cdots$ of labels of the shape (f,i) (in which case $arity(f) > 0$) or f (i.e. $f\,\epsilon$, in which case $arity(f) = 0$, and the label marks the end of the Σ-projection). We say the PDA $\mathcal{A}$ *generates* the Σ-labelled tree t just if the *branch language*[2] of t coincides with the set of Σ-projection of computation paths of $\mathcal{A}$.

Theorem 3 (Knapik et al. [8]). *There exists a reduction of an order-n tree-generating PDA $\mathcal{M}$ to an order-n safe recursion scheme $\mathcal{G}$ that generates the same tree as $\mathcal{M}$. Moreover, both the running time of the reduction algorithm and the size of $\mathcal{G}$ are polynomial in the size of $\mathcal{M}$.*

By Theorems 2 and 3, it suffices to show that, given a word w and an order-n APDA $\mathcal{A}$, one can construct an order-n tree-generating PDA $\mathcal{M}_{\mathcal{A},w}$ and a trivial APT $\mathcal{B}_{\mathcal{A}}$ such that w is accepted by $\mathcal{A}$ if, and only if, the tree generated by $\mathcal{M}_{\mathcal{A},w}$ is accepted by $\mathcal{B}_{\mathcal{A}}$.

Let w be a word over Σ. We write w_i ($i \in \{0,\ldots,|w|-1\}$) for the i-th element of w. From w and $\mathcal{A} = \langle P, \lambda, p_0, \Gamma, \Sigma, \Delta, F\rangle$ above, we construct an order-k PDA $\mathcal{M}_{\mathcal{A},w}$ for generating a $\{\mathtt{A},\mathtt{E},\mathtt{R},\mathtt{T}\}$-labelled tree, which expresses a kind of run tree of $\mathcal{A}$ over the input word w. The node label $\mathtt{A}$ ($\mathtt{E}$, resp) means that $\mathcal{A}$ is in a universal (existential, resp.) state; $\mathtt{T}$ means that $\mathcal{A}$ has accepted the word, and $\mathtt{R}$ means that $\mathcal{A}$ is stuck (having no outgoing transition).

Let $N := max_{q\in P, a\in\Sigma\cup\{\epsilon\},\gamma\in\Gamma}|\{q',\theta) \mid (q,\gamma,a,q',\theta)\in\Delta\}|$. I.e. N is the degree of non-determinacy of $\mathcal{A}$. We define $\mathcal{M}_{\mathcal{A},w} := \langle\{\mathtt{A},\mathtt{E},\mathtt{T},\mathtt{R}\},\Gamma,Q,\delta,(p_0,0)\rangle$ where:

– The ranked alphabet is $\{\mathtt{A},\mathtt{E},\mathtt{T},\mathtt{R}\}$, where the arities of $\mathtt{A}$ and $\mathtt{E}$ are N, and those of $\mathtt{T}$ and $\mathtt{R}$ are 0.

– $Q = (P\times\{0,\ldots,|w|\}) \cup \{q_\top, q_\bot\} \cup (P\times\{0,\ldots,|w|\}\times Op_n)$

– $\delta : Q\times\Gamma \longrightarrow (Q\times Op_n + \{(g;\widetilde{q}) : g\in\{\mathtt{A},\mathtt{E},\mathtt{T},\mathtt{R}\}, q_i\in Q\})$ is given by:

(1) $\delta((p,|w|),\gamma) = (\mathtt{T};\epsilon)$, if $p\in F$

(2) $\delta((p,i),\gamma) = (\mathtt{A};(p_1,j_1,\theta_1),\ldots,(p_m,j_m,\theta_m),\underbrace{q_\top,\ldots,q_\top}_{N-m})$
if $\lambda(p) = \mathtt{A}$ and $\{(p_1,j_1,\theta_1),\ldots,(p_m,j_m,\theta_m)\}$ is:
$\{(p',i+1,\theta) \mid (p,\gamma,w_i,p',\theta)\in\Delta \wedge i < |w|\} \cup \{(p',i,\theta) \mid (p,\gamma,\epsilon,p',\theta)\in\Delta\}$

(3) $\delta((p,i),\gamma) = (\mathtt{E};(p_1,j_1,\theta_1),\ldots,(p_m,j_m,\theta_m),q_\bot,\ldots,q_\bot)$
if $\lambda(p) = \mathtt{E}$ and $\{(p_1,j_1,\theta_1),\ldots,(p_m,j_m,\theta_m)\}$ is:
$\{(p',i+1,\theta) \mid (p,\gamma,w_i,p',\theta)\in\Delta \wedge i < |w|\} \cup \{(p',i,\theta) \mid (p,\gamma,\epsilon,p',\theta)\in\Delta\}$

(4) $\delta((p,i,\theta),\gamma) = ((p,i),\theta)$

(5) $\delta(q_\top,\gamma) = (\mathtt{T};\epsilon)$

(6) $\delta(q_\bot,\gamma) = (\mathtt{R};\epsilon)$

Rules (2) and (3) are applied only when rule (1) is inapplicable. $\mathcal{M}_{\mathcal{A},w}$ simulates $\mathcal{A}$ over the word w, and constructs a tree representing the computation

[2] The *branch language* of $t : dom(t) \longrightarrow \Sigma$ consists of (i) infinite words $(f_1,d_1)(f_2,d_2)\cdots$ just if there exists $d_1\,d_2\cdots\in\{1,2,\cdots,m\}^\omega$ (where m is the maximum arity of the Σ-symbols) such that $t(d_1\cdots d_i) = f_{i+1}$ for every $i\geq 0$; and (ii) finite words $(f_1,d_1)\cdots(f_n,d_n)\,f_{n+1}$ just if there exists $d_1\cdots d_n\in\{1,\cdots,m\}^*$ such that $t(d_1\cdots d_i) = f_{i+1}$ for $0\leq i\leq n$, and the arity of f_{n+1} is 0.

of $\mathcal{A}$. A state $(p, i) \in P \times \{0, \ldots, |w| - 1\}$ simulates $\mathcal{A}$ in state p reading the letter w_i. A state (p, i, θ) simulates an intermediate transition state of $\mathcal{A}$, where θ is the stack operation to be applied. The states $q_\top$ and $q_\bot$ are for creating dummy subtrees of nodes labelled with $\mathtt{A}$ or $\mathtt{E}$, so that the number of children of these nodes adds up to N, the arity of $\mathtt{A}$ and $\mathtt{E}$. Rule (1) ensures that when $\mathcal{A}$ has read the input word and reached a final state, $\mathcal{M}_{\mathcal{A},w}$ stops simulating $\mathcal{A}$ and outputs $\mathtt{T}$. Rule (2) is used to simulate transitions of $\mathcal{A}$ in a universal state, reading the i-th input: $\mathcal{M}_{\mathcal{A},w}$ constructs a node labelled $\mathtt{A}$ (to record that $\mathcal{A}$ was in a universal state) and spawns threads to simulate all possible transitions of $\mathcal{A}$. Rule (3) is for simulating $\mathcal{A}$ in an existential state. Note that, if $\mathcal{A}$ gets stuck (i.e. if there is no outgoing transition), all children of the $\mathtt{E}$-node are labelled $\mathtt{R}$; thus failure of the computation can be recognized by the trivial APT given in the following. Rule (4) is just for intermediate transitions. Note that a transition of $\mathcal{A}$ is simulated by $\mathcal{M}_{\mathcal{A},w}$ in two steps: the first for outputting $\mathtt{A}$ or $\mathtt{E}$, and the second for changing the stack.

Now, we construct a trivial APT that accepts the tree generated by $\mathcal{M}_{\mathcal{A},w}$ if, and only if, w is *not* accepted by $\mathcal{A}$. Let $\mathcal{B}_\mathcal{A}$ be $(\{q_0\}, \{\mathtt{A}, \mathtt{E}, \mathtt{T}, \mathtt{R}\}, q_0, \delta, \{q_0 \mapsto 0\})$ where:

$$\delta(q_0, \mathtt{A}) = \bigvee_{i=1}^{N}(i, q_0) \quad \delta(q_0, \mathtt{E}) = \bigwedge_{i=1}^{N}(i, q_0) \quad \delta(q_0, \mathtt{T}) = \mathtt{f} \quad \delta(q_0, \mathtt{R}) = \mathtt{t}$$

Intuitively, $\mathcal{B}_\mathcal{A}$ accepts all trees representing a failure computation tree of $\mathcal{A}$. If the automaton in state q_0 reads $\mathtt{T}$ (which corresponds to an accepting state of $\mathcal{A}$), it gets stuck. Upon reading $\mathtt{A}$, the automaton non-deterministically chooses one of the subtrees, and checks whether the subtree represents a failure computation of $\mathcal{A}$. On the other hand, upon reading $\mathtt{E}$, the automaton checks that all subtrees represent failure computation trees of $\mathcal{A}$.

Based on the above intuition, we can prove the following result.

Theorem 4. *Let w be a word, and $\mathcal{A}$ an order-n APDA. Then w is* not *accepted by $\mathcal{A}$ if, and only if, the tree generated by $\mathcal{M}_{\mathcal{A},w}$ is accepted by $\mathcal{B}_\mathcal{A}$.*

Corollary 1. *The model checking of an order-n recursion scheme with respect to a trivial APT is n-EXPTIME-hard in the size of the recursion scheme.*

By modifying the encoding, we can also show that the model checking problem is n-EXPTIME-hard in the size of APT. The idea is to modify $\mathcal{M}_{\mathcal{A},w}$ so that it generates a tree representing computation of $\mathcal{A}$ over not just w but all possible input words, and let a trivial APT check the part of the tree corresponding to the input word w. As a result, the trivial APT depends on the input word w, but the tree-generating PDA does not. See [7] for more details. To our knowledge, the lower-bound (of the complexity of model-checking recursion schemes) in terms of the size of APT has been unknown even for the entire class of APT.

4 Disjunctive APT and Complexity of Model Checking

A *disjunctive APT* is an APT whose transition function δ is disjunctive, i.e. δ maps each state to a positive boolean formula θ that contains only disjunctions

and no conjunctions, as given by the grammar $\theta ::= \mathtt{t} \mid \mathtt{f} \mid (i,q) \mid \theta \vee \theta$. Disjunctive APT can be used to describe path (or linear-time) properties of trees.

First we give a logical characterization of disjunctive APT as follows. Call $\mathcal{D}$ the following "disjunctive fragment" of the modal mu-calculus:

$$\phi, \psi ::= P_f \wedge \phi \mid Z \mid \phi \vee \psi \mid \langle i \rangle \phi \mid \nu Z.\phi \mid \mu Z.\phi$$

where f ranges over symbols in Σ, and i ranges over $\{1, \cdots, \mathit{arity}(\Sigma)\}$.

Proposition 2 (Equi-Expressivity). *The logic $\mathcal{D}$ and disjunctive APT are equivalent for defining possibly-infinite ranked trees. I.e. for every closed $\mathcal{D}$-formula, there is a disjunctive APT that defines the same tree language, and vice versa.*

Remark 1. (i) A disjunctive APT is a non-deterministic parity tree automaton. (ii) For defining languages of ranked trees, disjunctive APT are a proper subset of the *disjunctive formulas* in the sense of Walukiewicz and Janin [9]. For example, the disjunctive formula $(1 \to \{\mathtt{t}\}) \wedge (2 \to \{\mathtt{t}\})$ is not equivalent to any disjunctive APT.

In the rest of the section, we show that the model checking problem for order-n recursion schemes is $(n-1)$-EXPTIME complete for disjunctive APT.

4.1 Upper Bound

We sketch a proof of the following theorem, based on Kobayashi and Ong's type system for recursion schemes [6]. An alternative proof, based on variable profiles [1], will be given in a forthcoming journal version of this paper [7].

Theorem 5. *Let $\mathcal{G}$ be an order-n recursion scheme and $\mathcal{B}$ a disjunctive APT. It is decidable in $(n-1)$-EXPTIME whether $\mathcal{B}$ accepts the value tree $[\![\mathcal{G}]\!]$ from its root.*

In a recent paper [6], we constructed an intersection type system equivalent to the modal mu-calculus model checking of recursion schemes, in the sense that for every APT, there is a type system such that the tree generated by a recursion scheme is accepted by the APT if, and only if, the recursion scheme is typable in the type system. Thus, the model checking problem is reduced to a type checking problem. The main idea of the type system is to refine the tree type $\mathtt{o}$ by the states and priorities of an APT: the type q describes a tree that is accepted by the APT with q as the start state. The intersection type $(\theta_1, m_1) \wedge (\theta_2, m_2) \to q$, which refines the type $\mathtt{o} \to \mathtt{o}$, describes a tree function that takes an argument which has types θ_1 and θ_2, and returns a tree of type q.

The type checking algorithm presented in *ibid.* is n-EXPTIME in the combined size of the order-n recursion scheme and APT (precisely the complexity is $O(r^{1+\lfloor m/2 \rfloor} \mathbf{exp}_n((a\,|Q|\,m)^{1+\epsilon}))$ for $n \geq 2$, where r is the number of rules and a the largest prioirty of symbols in the scheme, m is the largest priority, $|Q|$ is the

number of states) The bottleneck of the algorithm is the number of (atomic) intersection types, where the set $\mathcal{T}(\kappa)$ of atomic types refining a simple type κ is inductively defined by: $\mathcal{T}(\mathtt{o}) = Q$ and $\mathcal{T}(\kappa_1 \to \kappa_2) = \{\bigwedge S \to \theta \mid \theta \in \mathcal{T}(\kappa_2), S \subseteq \mathcal{T}(\kappa_1) \times P\}$. where Q and P are the sets of states and priorities respectively.

According to the syntax of atomic types above, the number of atomic types refining a simple type of order-n is n-exponential in general. In the case of disjunctive APT, however, for each type of the form $\mathtt{o} \to \cdots \to \mathtt{o} \to \mathtt{o}$, we need to consider only atomic types of the form $\bigwedge S_1 \to \cdots \to \bigwedge S_k \to q$, where at most one of the S_i's is a singleton set and the other S_j's are empty. Intuitively, this is because a run-tree of a disjunctive APT consists of a single path, so that the run-tree visits only one of the arguments, at most once. In fact, we can show that, if a recursion scheme is typable in the type system for a disjunctive APT, the recursion scheme is typable in a restricted type system in which order-1 types are constrained as described above: this follows from the proof of completeness of the type system [6], along with the property of the accepting run-tree mentioned above. Thus, the number of atomic types is $k \times |Q| \times |P| \times |Q|$ (whereas for general APT, it is exponential). Therefore, the number of atomic types possibly assigned to a symbol of order n is $(n-1)$-exponential. By running the same type checking algorithm as *ibid.* (but with order-1 types constrained as above), order-n recursion schemes can be type-checked (i.e. model-checked) in $(n-1)$-EXPTIME.

4.2 Lower Bound

We show the lower bound by a reduction of the emptiness problem of the finite-word language accepted by an order-n (deterministic) PDA (which is $(n-1)$-EXPTIME complete [5]). From an order-n PDA $\mathcal{A}$, we can construct an order-n tree-generating PDA $\mathcal{M}_{\mathcal{A}}$, which simulates all possible input and ϵ-transitions of $\mathcal{A}$, and outputs $\mathtt{e}$ only when $\mathcal{A}$ reaches a final state: See the long version [7]. By a result of Knapik et al. [8], we can construct an equi-expressive order-n safe recursion scheme $\mathcal{G}$. By the construction, the finite word-language accepted by $\mathcal{A}$ is non-empty if, and only if, the value tree of $\mathcal{G}$ has a node labelled $\mathtt{e}$. Since the latter property can be expressed by a disjunctive APT, the problem of model-checking recursion schemes for disjunctive APT is $(n-1)$-EXPTIME hard. The problem is $(n-1)$-EXPTIME hard also in the size of the disjunctive APT: See [7] for more details.

4.3 Path Properties

The *path language* of a Σ-labelled tree t is the image of the map F, which acts on the elements of the branch language of t by "forgetting the argument positions" i.e. $F : (f_1, d_1)\,(f_2, d_2) \cdots \mapsto f_1\, f_2 \cdots$ and $F : (f_1, d_1) \cdots (f_n, d_n)\, f_{n+1} \mapsto f_1 \cdots f_n\, f_{n+1}^{\omega}$. For example, the path language of the tree $f\, a\, (f\, a\, b)$ is $\{f\, a^{\omega}, f\, f\, a^{\omega}, f\, f\, b^{\omega}\}$. Let $\mathcal{G}$ be a recursion scheme. We write $\mathcal{W}(\mathcal{G})$ for the *path language* of $[\![\mathcal{G}]\!]$. Thus elements of $\mathcal{W}(\mathcal{G})$ are infinite words over the alphabet Σ which is now considered unranked (i.e. arities of the symbols are forgotten).

Lemma 1. *Let $\mathcal{G}$ be an order-n recursion scheme. The following problems are $(n-1)$-EXPTIME complete.*

(i) $\mathcal{W}(\mathcal{G}) \cap \mathcal{L}(\mathcal{C}) \stackrel{?}{=} \emptyset$, where $\mathcal{C}$ is a non-deterministic *parity word automaton.*

(ii) $\mathcal{W}(\mathcal{G}) \stackrel{?}{\subseteq} \mathcal{L}(\mathcal{C})$, where $\mathcal{C}$ is a deterministic *parity word automaton.*

The decision problems REACHABILITY (i.e. whether $[\![\mathcal{G}]\!]$ has a node labelled by a given symbol `e`) and FINITENESS (i.e. whether $[\![\mathcal{G}]\!]$ is finite) are instances of Problem (i) of Lemma 1; hence they are in $(n-1)$-EXPTIME (the former is $(n-1)$-EXPTIME complete, by the proof of Section 4.2).

5 Application to Resource Usage Verification

Now we apply the result of the previous section to show that the resource usage verification problem is $(n-1)$-EXPTIME complete. The aim of resource usage verification is to check whether a program accesses each resource according to the resource specification. For example, consider the following program.

```
let rec g x = if b then close(x) else read(x); g(x) in
let r = open_in "foo" in g(r)
```

It opens a read-only file "`foo`", reads and closes it. For this program, the goal of the verification is to statically check that the file is eventually closed before the program terminates, and after it is closed, it is never read from or written to.

The resource usage verification problem was formalized by Igarashi and Kobayashi [3]. Kobayashi [2] recently showed that the problem is decidable for the simply-typed λ-calculus with recursion, generated from a base type of booleans, and augmented by resource creation/access primitives, by reduction to the model checking problem for recursion schemes.

Kobayashi [2] considered a language in CPS (continuation passing style), with only top-level function definitions of the form $F\ \widetilde{x} = e$, where e is given by:

$$e ::= \star \mid x \mid F \mid e_1 e_2 \mid \mathbf{If}{*}\ e_1\ e_2 \mid \mathbf{New}^L\ e \mid \mathbf{Acc}_a\ e_1\ e_2$$

The term $\star$ is the unit value. The term $\mathbf{If}{*}\ e_1\ e_2$ is a non-deterministic branch between e_1 and e_2. The term $\mathbf{New}^L\ e$ creates a fresh resource that should be used according to L, and passes it to e (thus, e is a function that takes a resource as an argument). Here, L is a regular language. The term $\mathbf{Acc}_a\ e_1\ e_2$ accesses the resource e_1 (where a is the name of the access primitive), and then executes e_2. For example, in the above program, the last line is expressed by $\mathbf{New}^{r*c}\ G$, and `close(x)` is expressed by $\mathbf{Acc}_c\ x\ k$ (where k is the continuation).

The language is simply typed; the two base types are **unit** for unit values and **R** for resources. The body of each definition must have type **unit** (in other words, resources cannot be used as return values; in that sense, programs are in CPS). The constants $\mathbf{If}{*}$, $\mathbf{New}^L$, and $\mathbf{Acc}_a$ are given the following types.

$$\mathbf{If}{*}{:}\mathbf{unit} \to \mathbf{unit} \to \mathbf{unit}, \mathbf{New}^L{:}(\mathbf{R} \to \mathbf{unit}) \to \mathbf{unit}, \mathbf{Acc}_a{:}\mathbf{R} \to \mathbf{unit} \to \mathbf{unit}$$

A program can be transformed to a recursion scheme that generates a tree representing all possible (resource-wise) access sequences of the program [2]. We just need to replace each function definition $F\,\widetilde{x} = e$ with the rewrite rule $F\,\widetilde{x} \to e$, and add the following rules

$$\begin{array}{lll} \mathbf{If}* \; x \; y \to \mathtt{br}\, x\, y & \mathbf{Acc}_a \; x \; k \to x\, a\, k & \star \to \mathtt{t}(\star) \\ \mathbf{New}^L k \to \mathtt{br}\,(\nu^L(kI))\,(kK) & I\, x\, k \to x\, k & K\, x\, k \to k \end{array}$$

Here, `br` is a terminal for representing non-deterministic choice. In the rule for $\mathbf{New}^L$, a fresh resource is instantiated to either I or K. This is a trick used to extract resource-wise access sequences, by tracing or ignoring the new resource in a non-deterministic manner: see Kobayashi's paper [2] for more explanation. The above transformation preserves types, except that **unit** and **R** are replaced by $\mathtt{o}$ and $(\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$ respectively.

Along with the transformation above, a tree automaton can be constructed that accepts the trees containing an *invalid* access sequence. The automaton just needs to focus on paths that contain a single occurrence of $\mathbf{New}^L$, and check whether, for every sequence s below $\mathbf{New}^L$, all prefixes of s are elements of $L \cdot \mathtt{t}^*$ (with `br` ignored). Thus, the automaton belongs to the class of disjunctive APT. (On the other hand, an automaton that accepts the complement, i.e. the set of trees containing only valid sequences, belongs to the class of trivial APT.)

We now show that the resource usage verification is $(n-1)$-EXPTIME complete, where n is the largest order of types in the source program. The base types **unit** and **R** have orders 0 and 1 respectively. We assume that each resource specification in the program is given as a deterministic finite state automaton. The lower-bound can be shown by reduction of the reachability problem of recursion schemes to the resource usage verification problem: we just transform each rule $F\,\widetilde{x} \to t$ into the function definition $F\,\widetilde{x} = t$, and replace the terminal `e` with $\mathbf{New}^{\{\epsilon\}}\mathit{Fail}$, where Fail is defined by $\mathit{Fail}\; x = \mathbf{Acc}_{\mathtt{fail}}\; x \; \star$. Since resource primitives occur only in the transformation of `e`, the order of the resulting resource usage program is the maximum of 3 and the order of the recursion scheme. Thus, the resource usage verification is $(n-1)$-EXPTIME hard for $n \geq 3$ (note that 3 is the lowest order of a closed program that creates a resource, since $\mathbf{New}^L$ has order 3).

Showing the upper-bound is a little tricky: since the resource type **R** of order 1 is transformed into the type $(\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$ of order 2, a source program of order-n may be transformed into a recursion scheme of order $n+1$. For the image of the resource type, however, it is sufficient to consider only two atomic types σ_I and σ_K, where $\sigma_I = \bigwedge_{q_1,q_2}((q_1 \to q_2) \to q_1 \to q_2)$ and $\sigma_K = \bigwedge_q((\bigwedge \emptyset) \to q \to q)$. Here, we have omitted priorities. Thus, although, for example, a type $\mathbf{R} \to \cdots \to \mathbf{R} \to \mathbf{unit}$ of order 2 is transformed into an order-3 type, the number of atomic types that should be considered is single-exponential. Since the APT for recognizing the value tree is disjunctive, we can apply the argument in Section 4 to conclude that the recursion scheme can be model-checked in $(n-1)$-EXPTIME.

Acknowledgments. We would like to thank the anonymous reviewers for useful comments. This work was partially supported by Kakenhi 20240001 and EPSRC EP/F036361.

References

1. Ong, C.-H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006, pp. 81–90. IEEE Computer Society Press, Los Alamitos (2006)
2. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, pp. 416–428 (2009)
3. Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Transactions on Programming Languages and Systems 27(2), 264–313 (2005)
4. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. Logical Methods in Computer Science 3(3) (2007)
5. Engelfriet, J.: Iterated stack automata and complexity classes. Information and Computation 95(1), 21–75 (1991)
6. Kobayashi, N., Ong, C.-H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009 (2009)
7. Kobayashi, N., Ong, C.-H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. A long version, available from the authors' web page (2009)
8. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)
9. Janin, D., Walukiewicz, I.: Automata for the modal mu-calculus and related results. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 552–562. Springer, Heidelberg (1995)