Automata-Based Abstraction Refinement for μHORS Model Checking

Naoki Kobayashi* and Xin Li†
The University of Tokyo
e-mail: *koba@is.s.u-tokyo.ac.jp, †li-xin@kb.is.s.u-tokyo.ac.jp

***Abstract*—The model checking of higher-order recursion schemes (HORS), aka. higher-order model checking, is the problem of checking whether the tree generated by a given HORS satisfies a given property. It has recently been studied actively and applied to automated verification of higher-order programs. Kobayashi and Igarashi studied an extension of higher-order model checking called μHORS model checking, where HORS has been extended with recursive types, so that a wider range of programs, including object-oriented programs and multi-threaded programs, can be precisely modeled and verified. Although the μHORS model checking is undecidable in general, they developed a sound but incomplete procedure for μHORS model checking. Unfortunately, however, their procedure was not scalable enough. Inspired by recent progress of (ordinary) HORS model checking, we propose a new procedure for μHORS model checking, based on automata-based abstraction refinement. We have implemented the new procedure and confirmed that it often outperforms the previous procedure.**

***Keywords*-higher-order model checking; tree automata; abstraction refinement**

I. Introduction

A higher-order recursion scheme (HORS) [1], [2] is a simply-typed, higher-order grammar for generating a possibly infinite tree. It can be considered a simply-typed functional program with recursion and tree constructors (but not destructors). The model checking of HORS is the problem of checking whether the tree generated by a given HORS satisfies a given tree property. HORS model checking can be considered a generalization of finite-state and pushdown model checking, and has been applied to automated verification of functional programs [3], [4], [5]. The idea of applying HORS model checking to program verification is to transform a given source program (possibly after predicate abstraction [4]) to a HORS that generates a tree representing all the possible execution sequences of the source program [3], so that the problem of checking a property of the source program is reduced to that of checking a property of the tree generated by the HORS. The advantage of using HORS model checking for program verification is that higher-order functional programs can be precisely modeled as HORS. If higher-order functional programs were modeled as finite state systems and finite-state model checking [6] were applied, a lot of control information would be lost by the modeling.

While HORS can serve as a precise model of a *simply-typed* higher-order functional program, it does not do so for programs of more expressive languages, such as functional programs with recursive types, object-oriented programs, and multi-threaded programs. To remedy the problem, Kobayashi and Igarashi [7] introduced μHORS, an extension of HORS with recursive types, and considered a model checking problem for μHORS. Thanks to recursive types, object-oriented programs and multi-threaded programs can be naturally modeled by μHORS. Although μHORS model checking is undecidable, they developed a sound procedure for μHORS model checking; furthermore, it is relatively complete with respect to recursive intersection types: if there exist recursive intersection types that serve as a certificate that μHORS satisfies a given property, then their procedure will eventually succeed. They have implemented the procedure and reported that it worked for small but non-trivial examples.

Kobayashi and Igarashi's previous algorithm for μHORS model checking, however, suffers from the following limitations:

1) It does not scale well, especially with respect to the size of the property automaton. Roughly, their procedure prepares a template of recursive intersection types, and looks for a proper instantiation of the template by using a SAT solver. When there is no solution, the template is expanded and the same procedure is repeated. With this approach, the size of SAT formulas quickly blows up.

2) Their procedure is not effective when a property is NOT satisfied. It just relies on an exhaustive search of the (possibly infinite) state space.

3) The relative completeness condition (the typability in a recursive intersection type system) is somehow specialized for their approach.

To remedy the problems above, we propose a new procedure for μHORS model checking. It is based on automata-based abstraction: each term that occurs during reductions is abstracted as a state of a tree automaton that accepts the term tree, and the automaton used for abstraction is gradually refined based on counterexamples. Our new algorithm has the following advantages over Kobayashi and Igarashi's previous procedure [7].

1) The abstraction refinement is more lightweight, and therefore, our procedure is expected to scale better, as confirmed by experiments.

2) When a property is not satisfied, our new procedure can find a counterexample more effectively.

3) The assumption for the relative completeness is replaced by a (arguably) more familiar condition that there exists a

regular invariant (a regular tree language that (i) is closed under reduction, (ii) contains the initial state, and (iii) contains no error state). Actually, we show that this assumption is equivalent to the assumption of the previous work based on the typability.

Technically, our procedure borrows some ideas of the two state-of-the-art model checking algorithms for (ordinary) HORS: HorSat [8] and Preface [9]. For constructing the initial abstraction, we use HorSat to collect intersection types of each symbol and convert them to tree automata. For abstract execution of μHORS, we construct a Preface-style abstract configuration graph. Unlike HorSat and Preface, however, our abstraction is automata-based; in fact, we cannot directly use intersection types to guarantee the relative completeness property mentioned above. For this reason, (despite the use of similar abstract configuration graphs) the mechanism of abstraction refinement is quite different from that of Preface.

We have implemented a prototype μHORS model checker and confirmed it often outperforms the previous procedure.

The rest of the paper is organized as follows. Section II reviews the definition of the μHORS model checking problem. Section III describes the new procedure for μHORS model checking. Section IV shows that the two assumptions for relative completeness are actually equivalent. Section V reports the implementation and experimental results. Section VI discusses related work and Section VII concludes the paper.

II. Preliminaries

A. μHORS and Model Checking Problem

The set of *types*, ranged over by κ, is defined by the syntax:

$$\kappa ::= \kappa_1 \to \cdots \to \kappa_m \to \mathsf{o} \mid \alpha \mid \mu\alpha.\kappa$$

Intuitively, o describes trees. The type $\kappa_1 \to \cdots \to \kappa_m \to \mathsf{o}$ (where m may be 0) describes a function that takes m arguments $x_1, \dots, x_m$ of types $\kappa_1, \dots, \kappa_m$ and returns a tree. The type $\mu\alpha.\kappa$ is a recursive type that satisfies $\alpha = \kappa$. The prefix $\mu\alpha$ binds α, and we call a type κ *closed* if all the type variables in κ are bound. We consider only closed types below. As usual [10], we consider that a (closed) recursive type $\mu\alpha.\kappa$ denotes a regular tree constructed from type constructors $\to$ and o, and identify two recursive types if their corresponding regular trees are identical. For example, we do not distinguish between $\mu\alpha.(\alpha \to \mathsf{o})$ and $\mu\alpha.((\alpha \to \mathsf{o}) \to \mathsf{o})$. We assume that $\to$ binds tighter than $\mu\alpha$, so that $\mu\alpha.\alpha \to \mathsf{o}$ denotes $\mu\alpha.(\alpha \to \mathsf{o})$, not $(\mu\alpha.\alpha) \to \mathsf{o}$. We sometimes call types *sorts*, following [7].

A *ranked alphabet* Σ is a map from a set of symbols (ranged over by a) to non-negative integers. A Σ*-labeled (ranked) tree* T is a map from a subset of $\{1, \dots, m\}^*$ to $dom(\Sigma)$, such that (i) $dom(T)$ is closed under the prefix operation, and (ii) $T(\pi) = a$ implies $\{i \mid \pi i \in dom(T)\} = \{1, \dots, \Sigma(a)\}$. Intuitively, each element a of a ranked alphabet serves as a tree constructor, and $\Sigma(a)$ denotes its arity (i.e., the number of children of each node labeled by a). For a map f, we write $f\{x \mapsto v\}$ for the map f' obtained by extending f with $f'(x) = v$.

Given a ranked alphabet Σ, the set of (applicative) terms is given by

$$t ::= x \mid a \mid t_1 t_2$$

where x ranges over a set of variables, and a ranges over $dom(\Sigma)$.

A type environment $\mathcal{K}$ is a map from a finite set of variables to the set of closed types. As usual, the type judgment relation $\mathcal{K} \vdash x : \kappa$ is defined by the following rules:

$$\frac{}{\mathcal{K}, x : \kappa \vdash x : \kappa} \qquad \frac{}{\mathcal{K} \vdash a : \underbrace{\mathsf{o} \to \cdots \to \mathsf{o}}_{\Sigma(a)} \to \mathsf{o}}$$

$$\frac{\mathcal{K} \vdash t_0 : \kappa_1 \to \kappa_2 \qquad \mathcal{K} \vdash t_1 : \kappa_1}{\mathcal{K} \vdash t_0 t_1 : \kappa_2}$$

When $\emptyset \vdash t : \mathsf{o}$, we identify t with the corresponding (finite) Σ-labeled ranked tree.

A μHORS [7] is a higher-order recursion scheme [2] extended with recursive types.

Definition 2.1 (μHORS): A μHORS is a quadruple $(\mathcal{N}, \Sigma, \mathcal{R}, S)$, where

- $\mathcal{N}$ is a map from a set of symbols (called *non-terminals*) to the set of types, where $\mathcal{N}(S) = \mathsf{o}$.
- Σ is a ranked alphabet. We call an element of $dom(\Sigma)$ a *terminal*.
- $\mathcal{R}$ is a set of rewriting rules of the form $F\,x_1 \cdots x_m \to t$ where F is a non-terminal of type $\kappa_1 \to \cdots \to \kappa_m \to \mathsf{o}$, and t is an applicative term such that $\mathcal{N}, x_1 : \kappa_1, \dots, x_m : \kappa_m \vdash t : \mathsf{o}$. For each non-terminal F, there must be exactly one rewriting rule.

A μHORS can be considered a higher-order, typed functional program for generating a (possibly infinite) tree (that has terminals as tree constructors). For a μHORS $\mathcal{G}$, we sometimes write $\mathcal{N}_\mathcal{G}, \Sigma_\mathcal{G}, \mathcal{R}_\mathcal{G}, S_\mathcal{G}$ for the four components of $\mathcal{G}$. The rewrite relation $\longrightarrow_\mathcal{G}$ on applicative terms is defined by

$$\frac{F\,x_1 \cdots x_m \to t \in \mathcal{R}_\mathcal{G}}{F\,s_1 \cdots s_m \longrightarrow_\mathcal{G} [s_1/x_1, \dots, s_m/x_m]t}$$

$$\frac{t_i \longrightarrow_\mathcal{G} t_i' \qquad \Sigma(a) = m}{a\,t_1 \cdots t_i \cdots t_m \longrightarrow_\mathcal{G} a\,t_1 \cdots t_i' \cdots t_m}$$

Let $\bot \notin dom(\Sigma)$ be a special symbol. For an applicative term of type o, we define the $\Sigma\{\bot \mapsto 0\}$-labeled ranked tree $t^\bot$ by

$$(a\,t_1 \cdots t_m)^\bot = a\,(t_1{}^\bot) \cdots (t_m{}^\bot) \qquad (F\,t_1 \cdots t_m)^\bot = \bot$$

The tree generated by $\mathcal{G}$, written $\mathbf{Tree}(\mathcal{G})$, is the $\Sigma\{\bot \mapsto 0\}$-labeled tree: $\bigsqcup\{t^\bot \mid S_\mathcal{G} \longrightarrow_\mathcal{G}^* t\}$. Here, $\bigsqcup$ is the least upper-bound with respect to the least partial order that satisfies $C[\bot] \sqsubseteq C[t]$ for any tree t and tree context C.

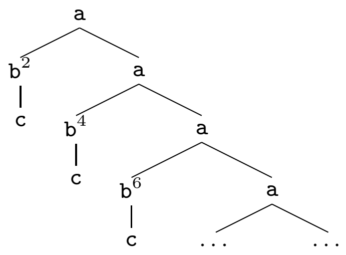
Example 2.1: Let $\mathcal{G}_1 = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where

$$\begin{aligned}
&\Sigma = \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0\}\\
&\mathcal{N} = \{S \mapsto \mathsf{o}, F \mapsto \mu\alpha.(\alpha \to (\mathsf{o} \to \mathsf{o}) \to \mathsf{o}),\\
&\qquad B \mapsto (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}\}\\
&\mathcal{R} = \{S \to F\,F\,\mathtt{b}, F\,f\,g \to \mathtt{a}\,(g(g\mathtt{c}))\,(f\,f\,(Bg)),\\
&\qquad B\,h\,x \to \mathtt{b}(h\,x)\}
\end{aligned}$$

S is reduced as follows:

$$S \to F\,F\,\mathtt{b} \to \mathtt{a}\,(\mathtt{b}^2\,\mathtt{c})\,(F\,F\,(B\,\mathtt{b})) \to \cdots.$$

The tree generated by $\mathcal{G}_1$ is as follows:



Definition 2.2: A *trivial tree automaton* is a quadruple $\mathcal{A} = (\Sigma, Q, \delta, Q_0)$, where Σ is a ranked alphabet, Q is a set of states, $\delta \subseteq Q \times dom(\Sigma) \times Q^*$ is a transition function such that $(q, a, q_1 \cdots q_m) \in \delta$ implies $m = \Sigma(a)$, and $Q_0 \subseteq Q$. A *run tree* of $\mathcal{A}$ over T is a Q-labeled (unranked) tree R such that (i) $dom(R) = dom(T)$, (ii) $R(\epsilon) \in Q_0$, and (iii) $(R(\pi), T(\pi), R(\pi 1) \cdots R(\pi\Sigma(a))) \in \delta$ for every $\pi \in dom(R)$. $\mathcal{A}$ *accepts* T if there is a run tree of $\mathcal{A}$ over T. If Q_0 is a singleton set $\{q_0\}$ and if there is at most one $q_1 \cdots q_m$ such that $(q, a, q_1 \cdots q_m) \in \delta$ for each q, a, the automaton is called *topdown-deterministic* (or just *deterministic*). If there is at most one q such that $(q, a, q_1 \cdots q_m) \in \delta$ for each $q_1 \cdots q_m, a$, the automaton is called *bottom-up deterministic*. We write $\mathcal{L}(\mathcal{A})$ for the set of trees accepted by $\mathcal{A}$, and write $\mathcal{L}(\mathcal{A}, q)$ for $\mathcal{L}(\mathcal{A}')$ where $\mathcal{A}'$ is the automaton obtained by replacing the initial state with q, i.e., $\mathcal{A}' = (\Sigma, Q, \delta, \{q\})$. The equivalence relation $\sim_{\mathcal{A}}$ on Σ-labeled trees is defined by: $T \sim_{\mathcal{A}} T' \overset{\text{def}}{\Leftrightarrow} \forall q \in Q.(T \in \mathcal{L}(\mathcal{A}, q) \Leftrightarrow T' \in \mathcal{L}(\mathcal{A}, q))$.

We often write $q \longrightarrow_\delta a\, q_1 \cdots q_n$ for $(q, a, q_1 \cdots q_n) \in \delta$, and omit the subscript δ. For a trivial automaton $\mathcal{A}$, we sometimes write $\Sigma_{\mathcal{A}}, Q_{\mathcal{A}}, \delta_{\mathcal{A}}, Q_{\mathcal{A},0}$ for the four components of $\mathcal{A}$. Note that, for finite trees, a topdown-deterministic trivial tree automaton is just an ordinary deterministic topdown tree automaton, and a bottom-up deterministic trivial tree automaton is just an ordinary deterministic bottom-up tree automaton [11].

For a tree automaton $\mathcal{A} = (\Sigma, Q, \delta, Q_0)$, we write $\mathcal{A}^\perp$ for the automaton $(\Sigma\{\perp \mapsto 0\}, Q, \delta \cup \{(q, \perp, \epsilon) \mid q \in Q\}, \delta, Q_0)$.

A *µHORS model checking problem* is the problem of, given a μHORS $\mathcal{G}$ and a trivial tree automaton $\mathcal{A}$, deciding whether $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$. Unfortunately, the μHORS model checking problem is undecidable [7] unlike the model checking of ordinary HORS [2]. For the sake of simplicity, in this paper, we consider only the case where $\mathcal{A}$ is topdown deterministic.

Example 2.2: Let $\mathcal{A}_1$ be $(\Sigma, \{q_0, q_1\}, \delta, \{q_0\})$ where Σ is as given in Example 2.1, and δ is as follows:

$$\{(q_0, \mathtt{a}, q_0 q_0), (q_1, \mathtt{a}, q_1 q_1), (q_0, \mathtt{b}, q_1), (q_1, \mathtt{b}, q_0), (q_0, \mathtt{c}, \epsilon)\}.$$

$\mathcal{A}_1$ accepts the trees such that every path from the root to a leaf contains an even number of $\mathtt{b}$'s. In particular, $\mathcal{A}_1$ accepts the tree generated by $\mathcal{G}_1$ in Example 2.1.

B. Invariants and Relative Completeness Criterion

This section gives some characterization of the μHORS model checking problem, and introduces the notion of regular invariants as the criterion for relative completeness.

The following fact is trivial by the definition of $\mathbf{Tree}(\mathcal{G})$ (see, e.g., [3]).

Fact 2.1: $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$ if and only if $\{t^\perp \mid S_{\mathcal{G}} \longrightarrow^*_{\mathcal{G}} t\} \subseteq \mathcal{L}(\mathcal{A}^\perp)$.

For a model checking problem $\mathbf{Tree}(\mathcal{G}) \overset{?}{\in} \mathcal{L}(\mathcal{A}^\perp)$, a set I of (applicative) terms is an *(inductive) invariant* if I satisfies the following conditions.

1) $S_{\mathcal{G}} \in I$
2) I is closed under reduction, i.e, if $t \in I$ and $t \longrightarrow_{\mathcal{G}} t'$, then $t' \in I$.
3) I contains no invalid tree, i.e., if $t \in I$, then $t^\perp \in \mathcal{L}(\mathcal{A}^\perp)$.

By Fact 2.1, there exists an inductive invariant if and only if $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$ holds.

A *regular* invariant is an inductive invariant I that is a regular (tree) language (i.e., accepted by some tree automaton). Our goal is to develop a μHORS model checking procedure that satisfies the following conditions:

1) If $\mathbf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^\perp)$, then the procedure eventually terminates and reports that the property is not satisfied (and outputs a reduction sequence $S_{\mathcal{G}} \longrightarrow^*_{\mathcal{G}} t$ such that $t^\perp \notin \mathcal{L}(\mathcal{A}^\perp)$).

2) If there exists a regular invariant, then the procedure eventually terminates and reports that the property is satisfied (and outputs some regular invariant).

A regular invariant can be expressed by a tree automaton that accepts term trees. When considering a term automaton below, we make the application symbol explicit. For example, we write $\mathtt{b}\,\mathtt{c}$ as $@_{\mathtt{o},\mathtt{o}}\mathtt{b}\,\mathtt{c}$ (where $@_{\kappa_1,\kappa_2}$ is considered a symbol of type $(\kappa_1 \to \kappa_2) \to \kappa_1 \to \kappa_2$). We use the two representations of a term (where @ is implicit or explicit) interchangeably.

Definition 2.3 (term automaton): A *term automaton* for μHORS $\mathcal{G}$ is a bottom-up deterministic tree automaton $\mathcal{B} = (\Sigma, \bigcup_{\kappa \in \mathit{Types}_{\mathcal{G}}} Q_\kappa, \delta, Q_0)$, where $\mathit{Types}_{\mathcal{G}}$ is the set of types occurring in $\mathcal{G}$, Q_0 is a subset of Q_κ for some κ, and

$$\begin{aligned}\Sigma = \{&@_{\kappa_1,\kappa_2} \mapsto 2 \mid \kappa_1 \to \kappa_2 \in \mathit{Types}_{\mathcal{G}}\} \\ &\cup \{a \mapsto 0 \mid a \in dom(\Sigma_{\mathcal{G}})\} \cup \{F \mapsto 0 \mid F \in dom(\mathcal{N}_{\mathcal{G}})\}\end{aligned}$$

The transition function δ should respect typing, i.e.,

- If $(q, @_{\kappa_1,\kappa_2}, q_0 q_1) \in \delta$, then $q \in Q_{\kappa_2}$, $q_0 \in Q_{\kappa_1 \to \kappa_2}$, and $q_1 \in Q_{\kappa_1}$.
- If $(q, a, \epsilon) \in \delta$, then $q \in Q_{\underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{\Sigma_{\mathcal{G}}(a)} \to \mathtt{o}}$.
- If $(q, F, \epsilon) \in \delta$, then $q \in Q_{\mathcal{N}_{\mathcal{G}}(F)}$.

We further require that δ is total (when it is interpreted as a bottom-up transition function), i.e., for any input symbol $e \in \Sigma$ of arity[1] n and $q_1 \cdots q_n$, there exists some q such that $(q, e, q_1 \cdots q_n) \in \delta$.

[1]By the definition of Σ above, n is actually either 0 or 2.

Let t be a term consisting of terminals, non-terminals, applications ($@_{\kappa_1,\kappa_2}$), and states of Q. We define the bottom-up transition relation $t \longrightarrow_{\mathcal{B}} t'$ by

$$C[e\, q_1 \cdots q_n] \longrightarrow_{\mathcal{B}} C[q] \text{ if } (q, e, q_1 \cdots q_n) \in \delta$$

where C is any context.

Example 2.3: Recall $\mathcal{G}_1$ in Example 2.1. Let $\mathcal{B}_0$ be $(\Sigma_0, Q, \delta, \{\xi_{\mathtt{o}}\})$ where

$$\begin{aligned}
&\Sigma_0 = \{@_{\kappa,(\mathtt{o}\to\mathtt{o})\to\mathtt{o}} \mapsto 2, @_{\mathtt{o}\to\mathtt{o},\mathtt{o}} \mapsto 2,\\
&\quad @_{\mathtt{o}\to\mathtt{o},\mathtt{o}\to\mathtt{o}} \mapsto 2, @_{\mathtt{o},\mathtt{o}\to\mathtt{o}} \mapsto 2, @_{\mathtt{o},\mathtt{o}} \mapsto 2,\\
&\quad S \mapsto 0, F \mapsto 0, B \mapsto 0, \mathtt{a} \mapsto 0, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 0\}\\
&Q = Q_\kappa \cup Q_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}} \cup Q_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}\\
&\quad \cup Q_{\mathtt{o}\to\mathtt{o}\to\mathtt{o}} \cup Q_{\mathtt{o}\to\mathtt{o}} \cup Q_{\mathtt{o}}\\
&Q_\kappa = \{\xi_\kappa\}, Q_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}} = \{\xi_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}}\}, Q_{\mathtt{o}\to\mathtt{o}\to\mathtt{o}} = \{\xi_{\mathtt{o}\to\mathtt{o}\to\mathtt{o}}\},\\
&Q_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} = \{\xi_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}\}, Q_{\mathtt{o}\to\mathtt{o}} = \{\xi_{\mathtt{o}\to\mathtt{o}}\}, Q_{\mathtt{o}} = \{\xi_{\mathtt{o}}\}\\
&\kappa = \mu\alpha.(\alpha \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o})
\end{aligned}$$

The transition rules are as follows:

$$\begin{array}{llll}
\xi_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}} \longrightarrow @\,\xi_\kappa\,\xi_\kappa & \xi_{\mathtt{o}} \longrightarrow @\,\xi_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}}\,\xi_{\mathtt{o}\to\mathtt{o}} & & \\
\xi_{\mathtt{o}\to\mathtt{o}} \longrightarrow @\,\xi_{\mathtt{o}\to\mathtt{o}\to\mathtt{o}}\,\xi_{\mathtt{o}} & \xi_{\mathtt{o}\to\mathtt{o}} \longrightarrow @\,\xi_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}\,\xi_{\mathtt{o}\to\mathtt{o}} & & \\
\xi_{\mathtt{o}} \longrightarrow @\,\xi_{\mathtt{o}\to\mathtt{o}}\,\xi_{\mathtt{o}} & \xi_\kappa \longrightarrow F & \xi_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} \longrightarrow B & \\
\xi_{\mathtt{o}} \longrightarrow S & \xi_{\mathtt{o}\to\mathtt{o}\to\mathtt{o}} \longrightarrow \mathtt{a} & \xi_{\mathtt{o}\to\mathtt{o}} \longrightarrow \mathtt{b} & \xi_{\mathtt{o}} \longrightarrow \mathtt{c}.
\end{array}$$

Here, we have omitted the subscript for @ for readability. Then, $t \longrightarrow^*_{\mathcal{B}_0} \xi_{\kappa'}$ if and only if t is a term of type κ'. In particular, $\mathcal{B}_0$ accepts all the terms of type $\mathtt{o}$. □

Example 2.4: Let $\mathcal{B}_1$ be $(\Sigma_0, Q, \delta, \{\xi_{\mathtt{acc}}\})$ where Σ_0 is as given in Example 2.3, and

$$\begin{aligned}
&Q = Q_\kappa \cup Q_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}} \cup Q_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} \cup Q_{\mathtt{o}\to\mathtt{o}\to\mathtt{o}}\\
&\quad \cup Q_{\mathtt{o}\to\mathtt{o}} \cup Q_{\mathtt{o}}\\
&Q_\kappa = \{\xi_F\}, Q_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}} = \{\xi_{FF}\}, Q_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} = \{\xi_B\},\\
&Q_{\mathtt{o}\to\mathtt{o}\to\mathtt{o}} = \{\xi_{\mathtt{a}}\}, Q_{\mathtt{o}\to\mathtt{o}} = \{\xi_{\mathtt{b}}, \xi_{B\mathtt{b}}, \xi_{\mathtt{ac}}\}, Q_{\mathtt{o}} = \{\xi_{\mathtt{acc}}, \xi_{\mathtt{c}}, \xi_{\mathtt{bc}}\}
\end{aligned}$$

$$\begin{array}{llll}
\xi_{F\,F} \longrightarrow @\,\xi_F\xi_F & \xi_{\mathcal{B}\mathtt{b}} \longrightarrow @\,\xi_B\xi_{\mathtt{b}} & \xi_{\mathtt{b}} \longrightarrow @\,\xi_B\xi_{B\mathtt{b}} & \\
\xi_{\mathtt{ac}} \longrightarrow @\,\xi_{\mathtt{a}}\xi_{\mathtt{c}} & \xi_{\mathtt{c}} \longrightarrow @\,\xi_{\mathtt{b}}\xi_{\mathtt{bc}} & \xi_{\mathtt{c}} \longrightarrow @\,\xi_{B\mathtt{b}}\xi_{\mathtt{c}} & \\
\xi_{\mathtt{bc}} \longrightarrow @\,\xi_{B\mathtt{b}}\xi_{\mathtt{bc}} & \xi_{\mathtt{bc}} \longrightarrow @\,\xi_{\mathtt{b}}\xi_{\mathtt{c}} & \xi_{\mathtt{acc}} \longrightarrow @\,\xi_{FF}\xi_{\mathtt{b}} & \\
\xi_{\mathtt{acc}} \longrightarrow @\,\xi_{FF}\xi_{B\mathtt{b}} & \xi_{\mathtt{acc}} \longrightarrow @\,\xi_{\mathtt{ac}}\xi_{\mathtt{acc}} & \xi_{\mathtt{bc}} \longrightarrow @\,\xi_{\mathtt{b}}\xi_{\mathtt{c}} & \\
\xi_{\mathtt{c}} \longrightarrow @\,\xi_{B\mathtt{b}}\xi_{\mathtt{c}} & \xi_F \longrightarrow F & \xi_B \longrightarrow B & \xi_{\mathtt{acc}} \longrightarrow S\\
\xi_{\mathtt{a}} \longrightarrow \mathtt{a} & \xi_{\mathtt{b}} \longrightarrow \mathtt{b} & \xi_{\mathtt{c}} \longrightarrow \mathtt{c} &
\end{array}$$

Here, we have omitted states that are not reachable to $\xi_{\mathtt{acc}}$; δ can be made total by adding dummy states. $\mathcal{B}_1$ accepts terms of the form $t_1(\cdots(t_n(F\,F(B^m\mathtt{b})))\cdots)$ or $t_1(\cdots(t_nS)\cdots)$ where each t_i is a term accepted from $\xi_{\mathtt{ac}}$. $\mathcal{L}(\mathcal{B}_1)$ is a regular invariant for $\mathbf{Tree}(\mathcal{G}_1) \in \mathcal{L}(\mathcal{A}_1{}^\perp)$, where $\mathcal{A}_1$ is the automaton in Example 2.2. □

We give another characterization of the μHORS model checking problem. For a μHORS $\mathcal{G}$ and a topdown-deterministic automaton $\mathcal{A}$, we define the relation $\longrightarrow_{\mathcal{G},\mathcal{A}}$ by the following rules:

- $(F\,t_1 \cdots t_m, q) \longrightarrow_{\mathcal{G},\mathcal{A}} ([t_1/x_1, \ldots, t_m/x_m]s, q)$ if $F\,x_1 \cdots x_m \to s \in \mathcal{R}_{\mathcal{G}}$.
- $(a\,t_1 \cdots t_m, q) \longrightarrow_{\mathcal{G},\mathcal{A}} (t_i, q_i)$ if $(q, a, q_1 \cdots q_m) \in \delta_{\mathcal{A}}$ and $i \in \{1, \cdots, m\}$.
- $(a\,t_1 \cdots t_m, q) \longrightarrow_{\mathcal{G},\mathcal{A}}$ `fail` if there exists no $q_1 \cdots q_m$ such that $(q, a, q_1 \cdots q_m) \in \delta_{\mathcal{A}}$.

Then, we have the following fact:

Fact 2.2: Let $\mathcal{G}$ be a μHORS and $\mathcal{A}$ a topdown-deterministic automaton, with $Q_{\mathcal{A},0} = \{q_0\}$. Then, $(S_{\mathcal{G}}, q_0) \longrightarrow^*_{\mathcal{G},\mathcal{A}}$ `fail` if and only if $\mathbf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^\perp)$.

The abstract configuration graph introduced in the next section can be considered an over-approximation of the reduction relation $\longrightarrow_{\mathcal{G},\mathcal{A}}$.

We give below an example of reduction from a program verification problem to a model checking problem for μHORS. An interested reader may wish to consult [3], [7] for a more detailed account of reduction from program verification problems to higher-order model checking.

Example 2.5: Consider the following OCaml-like program:

```
let mk_list b = cons b (cons (not b) nil)
let rec for_some p l =
  match l with
    [] -> false
  | x::l' -> (p x) || (for_some p l')
let main() =
  let b = * in
   assert(for_some (fun x->x) (mk_list b))
```

Here, $*$ represents a non-deterministic Boolean value. The main function first creates a non-deterministic Boolean value, binds `b` to it, makes a list of Booleans by calling `mk_list`, and then asserts that the list contains **true**.

Suppose we wish to check that the assertion in the program above never fails. Then we transform the program to the following μHORS $\mathcal{G}_2$, which generates the tree consisting of `br`, `end`, and `fail` that respectively denotes a non-deterministic branch, a successful termination, and an assertion failure.

$$\begin{array}{l}
\mathtt{Main} \to \mathtt{br}\ (C_1\,\mathtt{True})\ (C_1\,\mathtt{False})\\
C_1\,b \to \mathtt{Mklist}\,b\,C_2 \quad C_2\,l \to \mathtt{Forsome}\,\mathtt{Id}\,l\,\mathtt{Assert}\\
\mathtt{Assert}\,x \to \mathtt{If}\ x\,\mathtt{end}\,\mathtt{fail}\\
\mathtt{Mklist}\,b\,k \to k(\mathtt{Cons}\,b\ (\mathtt{Cons}\ (\mathtt{Not}\ b)\ \mathtt{Nil}))\\
\mathtt{Forsome}\,p\ l\ k \to l\ (k\,\mathtt{False})\ (C_3\ p\ k)\\
C_3\ p\ k\ x\ l \to \mathtt{If}\ (p\ x)\ (k\ \mathtt{True})\ (\mathtt{Forsome}\,p\ l\ k)\\
\mathtt{Id}\,x\,k_1\,k_2 \to x\ k_1\ k_2 \qquad \mathtt{If}\ b\ x\ y \to b\ x\ y\\
\mathtt{True}\,x\ y \to x \quad \mathtt{False}\,x\ y \to y \quad \mathtt{Not}\ x\ k_1\ k_2 \to x\ k_2\ k_1\\
\mathtt{Nil}\ k_1\ k_2 \to k_1 \qquad \mathtt{Cons}\ x\ l\ k_1\ k_2 \to k_2\ x\ l
\end{array}$$

Here, `Main` is the start symbol. The μHORS $\mathcal{G}_2$ above is essentially a CPS (continuation-passing-style [12]) version of the source program, where Booleans and lists are encoded based on Church encoding. The grammar first non-deterministically chooses a Boolean, and then calls `Mklist`, `Forsome`, and `Assert` in this order. The source program above may fail if and only if the tree generated by $\mathcal{G}_2$ contains `fail`. Thus, the verification problem has been reduced to a μHORS model checking problem. □

III. Model Checking Procedure

A. Overall Procedure

Fig. 1 shows the overall procedure $MC(\mathcal{G}, \mathcal{A})$. First, a term automaton $\mathcal{B}_0$ used for the initial abstraction is prepared (line 1). For relative completeness, $\mathcal{B}_0$ may be any automaton, although the efficiency of the procedure may often depend on the choice of $\mathcal{B}_0$. The automaton $\mathcal{B}$ used for abstraction is

```
1: $\mathcal{B}_0$ := initial automaton for abstraction;
2: $\mathcal{B}$ := $\mathcal{B}_0$;
3: Constr := $\emptyset$;
4: while true do {
5:   $\mathcal{C}$ := $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$;
6:   if $\mathcal{C}$ contains no error
7:   then return SATISFIED
8:   else if a real error is found then
9:          return UNSATISFIED
10:  else {
11:    $CE$ := a (spurious) counterexample;
12:    Constr := Constr$\cup${gen_constr($CE$)};
13:    $\mathcal{B}$ := refinement($\mathcal{B}_0$, Constr); } }
```

Fig. 1. Overview of the Model Checking Procedure $MC(\mathcal{G}, \mathcal{A})$.

initialized to $\mathcal{B}_0$, and the set Constr, which accumulates a set of constraints on $\mathcal{B}$, is initialized to the empty set (lines 2-3). After the initialization, the procedure enters an abstraction-refinement loop. In the loop, the μHORS $\mathcal{G}$ is abstractly reduced using the current automaton $\mathcal{B}$ for abstraction, in the style of the abstract configuration graph [9] (line 5). If no error is found in the abstract reduction, then we can conclude that the property is satisfied (i.e., $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$) (line 7). If there is an abstract error path, then we inspect it, and if it is a real one (i.e., if there is a corresponding concrete reduction sequence that generates an invalid tree), then it is reported that the property is violated (line 9). Otherwise, we pick a spurious counterexample CE, and add to Constr a new constraint gen_constr(CE) for avoiding CE (lines 11-12). We then find a new automaton $\mathcal{B}$ that is a refinement of $\mathcal{B}_0$ and satisfies Constr (lines 13). We repeat this abstraction-refinement loop until the property is proved or disproved. Note that, since the model checking problem is undecidable, the procedure may not terminate, repeating the abstraction-refinement loop forever.

In the rest of this section, we first explain lines 5–9 in Section III-B. We then explain the refinement procedure (lines 11-13) in Section III-C and the initial automaton construction in Section III-D. Finally, we discuss the properties of the procedure in Section III-E.

B. Automata-based Abstract Configuration Graph

Given a μHORS $\mathcal{G}$, a (top-down deterministic) tree automaton $\mathcal{A}$, and a term automaton $\mathcal{B}$ (recall Definition 2.3), an *abstract configuration graph* (ACG) is constructed as described below. A node in the ACG is either fail or of the form (t, q) where q is a state of $\mathcal{A}$ and t is an applicative term given by the syntax:

$$t ::= a \mid F \mid x^{\xi,\ell} \mid t_1 t_2.$$

Here, each variable is annotated with a state ξ of $\mathcal{B}$, and a label ℓ that uniquely identifies the corresponding binding. The existence of a node (t, q) intuitively means that every term represented by t is expected to generate a tree that is accepted by $\mathcal{A}$ from q. Besides the graph, a binding function ρ is constructed, which maps each variable $x^{\xi,\ell}$ to an applicative term. An ACG is constructed as follows.

1) Add the initial node (S, q_0) (where q_0 is the initial state of $\mathcal{A}$), and let the binding function ρ be the empty map.

2) Repeat the following procedures, until no more node or edge is added. When adding a node, we ignore the label part (denoted by ℓ), and merge the node with an existing one if the two nodes are identical up to label renaming.

- If there is a node $N = (F\, s_1 \cdots s_m, q)$, and $F\, x_1 \cdots x_m \rightarrow t \in \mathcal{R}_{\mathcal{G}}$ with $s_i \longrightarrow^*_{\mathcal{B}} \xi_i$ for each $i \in \{1, \ldots, m\}$ (here, we extend $\longrightarrow_{\mathcal{B}}$ by the rule $C[x^{\xi,\ell}] \longrightarrow_{\mathcal{B}} C[\xi]$), then we add a node $N' = ([x_1^{\xi_1,\ell_1}/x_1, \ldots, x_m^{\xi_m,\ell_m}/x_m]t, q)$ and an edge from N to N', where $\ell_1, \ldots, \ell_m$ are fresh labels. Also, extend ρ with $\rho(x_i^{\xi_i,\ell_i}) = s_i$.
- If there is a node $N = (a\, s_1 \cdots s_m, q)$ and $(q, a, q_1 \cdots q_m) \in \delta_{\mathcal{A}}$, then we add a node $N_i = (s_i, q_i)$ and an edge from N to N_i for each $i \in \{1, \ldots, m\}$. If there is no such $q_1 \cdots q_m$, then add a special node $N' =$ fail, and an edge from N to N'.
- If there is a node $N = (x^{\xi,\ell}\, s_1 \cdots s_m, q)$ and $\rho(x^{\xi,\ell'}) = t$ for some ℓ', then we add a node $N' = (t\, s_1 \cdots s_m, q)$ and add an edge labeled by (ℓ, ℓ') from N to N'.

We call an ACG graph *closed* if no more node or edge can be added in step 2, and denote it by $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$

Note that $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ always exists and is finite; furthermore, it is unique up to isomorphism and a renaming of labels. The finiteness of $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ follows from the fact that there can be finitely many nodes (up to label renaming), as in [9]: whenever a node (t, q) belongs to an ACG, t is of the form $t_1 \cdots t_k$ where each t_i is a term obtained from a term occurring in $\mathcal{G}$ by annotating each variable with a state of $\mathcal{B}$ and a label.

An ACG is considered an abstraction of the configuration graph introduced in [13], [3]. In the procedure above, the abstraction is performed by (i) merging nodes that are identical up to a renaming of labels, and (ii) introducing the edge (ℓ_1, ℓ_2) in the last case, where the variable with label ℓ_1 is instantiated to the value of the variable (of the same name and state) with a possibly different label ℓ_2.

Remark 3.1: A careful reader may notice that the initial states of $\mathcal{B}$ do not matter to the abstraction procedure above; what matters is only $\mathcal{L}(\mathcal{B}, q)$ for each q. Thus, we ignore the initial states of $\mathcal{B}$ below.

The following theorem states that the abstraction is sound in the sense that if the abstract reduction does not fail, then the answer to the model checking problem is yes.

Theorem 3.1: $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ exists (i.e., the step 2 above eventually terminates). If $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ does not contain the special node fail, then $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$.

Proof: We show the contraposition. Suppose that $\mathbf{Tree}(\mathcal{G})$ is not accepted by $\mathcal{A}$. Let *CCG* be the "concrete" (possibly infinite) configuration graph constructed in the same way as the ACG, except that (i) nodes are NOT identified up to label renaming, and (ii) the edge (ℓ_1, ℓ_2) is added only when $\ell_1 = \ell_2$. From Fact 2.2, it follows immediately that *CCG* contains fail. By the construction of the ACG, $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ also contains fail. ■

On the other hand, if $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ contains fail, then either (i) $\mathbf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^{\perp})$, or (ii) $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$ but the abstraction is too coarse. To find which is the case, we pick each error path of $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ from (S, q_0) to fail of up to a certain length (this is the number of edges of CE chosen in Section III-C), and check whether there is a corresponding concrete reduction sequence. If that is the case, we can conclude that $\mathbf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^{\perp})$, and output the corresponding concrete reduction sequence as a counterexample. Otherwise, we proceed to abstraction refinement as described in Section III-C. The procedure for checking whether there is a concrete reduction sequence corresponding to an abstract error path is more or less standard: from the error path, we extract a sequence consisting of elements of the form (a, i) (which means that the i-th branch was chosen at node $(a\, t_1 \cdots t_n, q))$ or F (which means that the non-terminal F has been reduced). Then we just need to check whether there is a concrete reduction sequence $(S, q_0) \longrightarrow^* \texttt{fail}$ that takes the same branch and reduces the same non-terminal at each step.

Example 3.1: Recall $\mathcal{G}_1$ in Example 2.1, $\mathcal{A}_1$ in Example 2.2, and $\mathcal{B}_0$ in Example 2.3. Fig. 2 shows (a part of) $\mathcal{C}_{\mathcal{G}_1,\mathcal{A}_1,\mathcal{B}_0}$, where the state annotations on variables are omitted. The bindings generated by the part of the ACG are as follows:

$$\begin{array}{ll}
\rho(f^{\xi_\kappa,[1]}) = \rho(f^{\xi_\kappa,[3]}) = F & \\
\rho(g^{\xi_{o\to o},[2]}) = \texttt{b} & \rho(g^{\xi_{o\to o},[4]}) = B\, g^{\xi_{o\to o},[2]} \\
\rho(h^{\xi_{o\to o},[5]}) = \rho(h^{\xi_{o\to o},[7]}) = g^{\xi_{o\to o},[2]} & \\
\rho(x^{\xi_{o\to o},[6]}) = \texttt{c} & \rho(x^{\xi_{o\to o},[8]}) = g^{\xi_{o\to o},[2]}\, \texttt{c}
\end{array}$$

The bindings for $f^{\xi_\kappa,[3]}$ and $g^{\xi_{o\to o},[4]}$ are created by the node $(F\ f^{[1]}\ (B\, g^{[2]}), q_0)$ in the figure; the resulting node $(\texttt{a}\,(g^{[4]}(g^{[4]}\,\texttt{c}))\,(f^{[3]}\,f^{[3]}\cdots), q_0)$ has been merged with the existing node $(\texttt{a}\,(g^{[2]}(g^{[2]}\,\texttt{c}))\,(f^{[1]}\,f^{[1]}\cdots), q_0)$ since they are equal up to label renaming. In the figure, the label (ℓ_1, ℓ_2) of an edge is shown only when $\ell_1 \neq \ell_2$; such an edge indicates that a binding different from the actual binding has been used in the reduction (so that a spurious error path may be created). In fact, the graph contains a path from (S, q_0) to fail, although $\mathbf{Tree}(\mathcal{G}_1)$ is accepted by $\mathcal{A}_1$. This is due to the edge labeled by $([2], [4])$, where $g^{\xi_{o\to o},[2]}$ has been instantiated to $\rho(g^{\xi_{o\to o},[4]}) = B\, g^{\xi_{o\to o},[2]}$ despite that the correct value for $g^{\xi_{o\to o},[2]}$ is $\rho(g^{\xi_{o\to o},[2]}) = \texttt{b}$. The dashed line is explained in Example 3.2. □

C. Abstraction Refinement

Given an ACG $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$, we compute a set CE of nodes, edges, labels, and label pairs such that

- CE includes (the nodes and edges in) a path from (S, q_0) to fail.
- If an edge labeled with (ℓ, ℓ') such that $\ell \neq \ell'$ belongs to CE, then $\ell, \ell', (\ell, \ell') \in CE$.
- If $\ell \in CE$ and $\rho(x^{\xi,\ell}) = s$, then CE should include a path from (S, q_0) to the node that has created the bindings $\rho(x^{\xi,\ell}) = s$, and all the labels in s should also belong to CE.

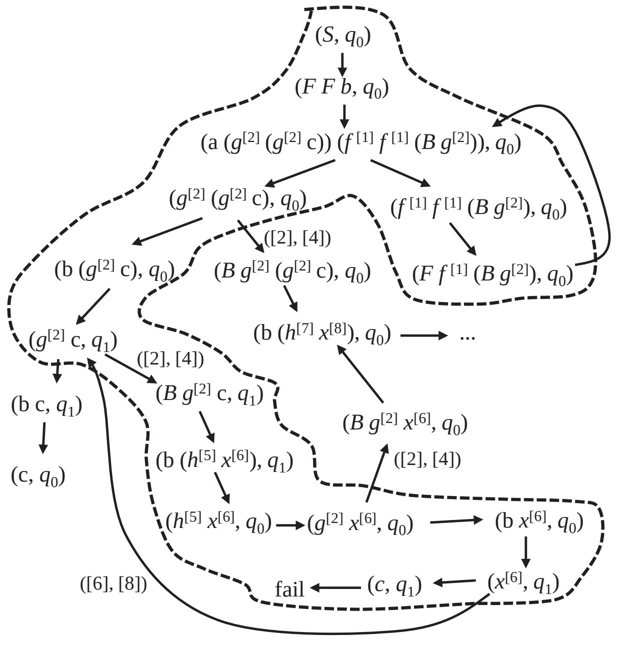


Fig. 2. Abstract configuration graph $\mathcal{C}_{\mathcal{G}_1,\mathcal{A}_1,\mathcal{B}_0}$.

Among CE's, we pick one such that the number of edges is minimal. For each label ℓ occurring in CE, we define a term $\rho^+(\ell)$ by

$$\rho^+(\ell) = [\rho^+(\ell_1)/x_1^{\xi_1,\ell_1}, \ldots, \rho^+(\ell_n)/x_n^{\xi_n,\ell_n}]s$$

if $\rho(x^{\xi,\ell}) = s$ for some x, ξ, where $x_1^{\xi_1,\ell_1}, \ldots, x_n^{\xi_n,\ell_n}$ are the variables occurring in s. Note that, by the construction of the abstract configuration graph, x, ξ, and s such that $\rho(x^{\xi,\ell}) = s$ are uniquely determined for each ℓ, and the above equality cannot be circular. Therefore, $\rho^+(\ell)$ is well defined. Then the constraint is as follows:

$$\bigvee_{(\ell,\ell') \in CE} \delta_{\mathcal{B}}(\rho^+(\ell)) \neq \delta_{\mathcal{B}}(\rho^+(\ell')).$$

The constraint means that we should refine the automaton $\mathcal{B}$ so that for some (ℓ, ℓ'), t^ℓ and $t^{\ell'}$ should be accepted by different states (i.e., $\rho^+(\ell) \longrightarrow^*_{\mathcal{B}} \xi$ and $\rho^+(\ell') \longrightarrow^*_{\mathcal{B}} \xi'$ imply $\xi \neq \xi'$). That ensures that in the next refinement step, CE is no longer detected as a counterexample.

Example 3.2: Recall the ACG shown in Fig. 2. The region enclosed by the dashed line shows a CE. The only label pair in CE is $([2], [4])$, hence the constraint is $\delta_{\mathcal{B}}(\rho^+([2])) \neq \delta_{\mathcal{B}}(\rho^+([4]))$, i.e.,

$$\delta_{\mathcal{B}}(\texttt{b}) \neq \delta_{\mathcal{B}}(B\,\texttt{b}).$$

This indicates that in the next step, the automaton $\mathcal{B}$ should distinguish between b and $B\,$b. □

Remark 3.2: In the feasibility check (to check whether there is a concrete reduction sequence that corresponds to an abstract error path) discussed at the end of Section III-B, we check all the abstract error paths shorter than the number of edges of CE chosen above. Then, the number of nodes of CE

monotonically increases as the refinement proceeds (because the number of constraints that may be generated by CE with a fixed number of nodes is bounded, and the same constraint is never encountered) we can ensure that a concrete error path is eventually found if there is any. $\square$

As a candidate of an updated automaton $\mathcal{B}$, we consider only a *refinement* of the initial automaton $\mathcal{B}_0$ in the following sense.

Definition 3.1: Let $\mathcal{B}_0 = (\Sigma, Q, \delta_{\mathcal{B}_0}, Q_0)$ be a term automaton and k a positive integer. A term automaton $\mathcal{B} = (\Sigma, Q', \delta_{\mathcal{B}}, Q'_0)$ is a *k-refinement* (or just a refinement) of $\mathcal{B}_0$ if the following holds:

- $Q' = Q \times \{1, \dots, k\}$
- $((q, i), a, (q_1, i_1) \cdots (q_n, i_n)) \in \delta_{\mathcal{B}}$ implies $(q, a, q_1 \cdots q_n) \in \delta_{\mathcal{B}_0}$
- $Q'_0 = Q_0 \times \{1, \dots, k\}$.

By the definition above, a set of (total) functions

$$\{f_{q,a,q_1 \cdots q_n} \in \{1, \dots, k\}^n \to \{1, \dots, k\} \\ \mid (q, a, q_1 \cdots q_n) \in \delta_{\mathcal{B}_0}\}$$

uniquely determines a k-refinement of $\mathcal{B}_0$, whose transition function is given as follows:

$$\delta_{\mathcal{B}} = \{((q, f_{q,a,q_1 \cdots q_n}(i_1, \dots, i_n)), a, (q_1, i_1) \cdots (q_n, i_n)) \\ \mid (q, a, q_1 \cdots q_n) \in \delta_{\mathcal{B}_0}, i_1, \dots, i_n \in \{1, \dots, k\}\}.$$

Thus, the constraint

$$\bigvee_{(\ell, \ell') \in CE} \delta_{\mathcal{B}}(\rho^+(\ell)) \neq \delta_{\mathcal{B}}(\rho^+(\ell'))$$

can be reduced to the constraint on functions $f_{q,a,q_1 \cdots q_n}$ as follows. First, annotate each constructor (which is an application $@_{\kappa_1,\kappa_2}$, a terminal a, or a non-terminal F) of the term trees $\rho^+(\ell)$ and $\rho^+(\ell')$ with the transition rule used in the sequence $\rho^+(\ell) \longrightarrow^*_{\mathcal{B}_0} q$ or $\rho^+(\ell') \longrightarrow^*_{\mathcal{B}_0} q$. Then, replace each constructor $e_{(q,e,q_1 \cdots q_n)}$ by the function symbol $f_{q,e,q_1 \cdots q_n}$. This gives us a constraint of the following form:

$$\bigvee_i u_i \neq u'_i,$$

where each of u_i, u'_i is an expression consisting of uninterpreted function symbols. Add this constraint to `Constr`. The satisfiability of `Constr` can be checked by an off-the-shelf SMT solver, like Z3. If `Constr` is satisfiable, an SMT solver returns a solution. Thus, we can use it to construct a new k-refinement $\mathcal{B}$. Otherwise, we increase the value of k until `Constr` becomes satisfiable.

Example 3.3: Recall Example 3.2. The constraint $\delta_{\mathcal{B}}(\mathtt{b}) \neq \delta_{\mathcal{B}}(B\,\mathtt{b})$ can be converted to the following formula on uninterpreted function symbols.

$$f_{\xi_{\mathtt{o}\to\mathtt{o}},\mathtt{b},\epsilon} \neq f_{\xi_{\mathtt{o}\to\mathtt{o}},@_{\mathtt{o}\to\mathtt{o},\mathtt{o}\to\mathtt{o}},\xi_{\kappa_1}\xi_{\mathtt{o}\to\mathtt{o}}}(f_{\xi_{\kappa_1},B,\epsilon}, f_{\xi_{\mathtt{o}\to\mathtt{o}},\mathtt{b},\epsilon})$$

where $\kappa_1 = (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$. This indicates that in the next step, the automaton $\mathcal{B}$ should distinguish between $\mathtt{b}$ and $B\,\mathtt{b}$. For $k = 2$, a solution for the formula above is as follows:

$$f_{\xi_{\mathtt{o}\to\mathtt{o}},\mathtt{b},\epsilon} = f_{\xi_{\kappa_1},B,\epsilon} = 1 \qquad f_{\xi_{\mathtt{o}\to\mathtt{o}},@_{\mathtt{o}\to\mathtt{o},\mathtt{o}\to\mathtt{o}},\xi_{\kappa_1}\xi_{\mathtt{o}\to\mathtt{o}}}(1,1) = 2 \\ f_{\xi_{\mathtt{o}\to\mathtt{o}},@_{\mathtt{o}\to\mathtt{o},\mathtt{o}\to\mathtt{o}},\xi_{\kappa_1}\xi_{\mathtt{o}\to\mathtt{o}}}(1,2) = 1$$

Then, the new automaton $\mathcal{B}$ accepts $\mathcal{B}^{2n}\mathtt{b}$ and $B^{2n+1}\mathtt{b}$ with states $(\xi_{\mathtt{o}\to\mathtt{o}}, 1)$ and $(\xi_{\mathtt{o}\to\mathtt{o}}, 2)$ respectively. With the refined automaton, $g^{\xi_{\mathtt{o}\to\mathtt{o}},[2]}$ and $g^{\xi_{\mathtt{o}\to\mathtt{o}},[4]}$ in Fig. 2 are refined to $g^{(\xi_{\mathtt{o}\to\mathtt{o}},1),[2]}$ and $g^{(\xi_{\mathtt{o}\to\mathtt{o}},2),[4]}$; thus, the bindings for $g^{(\xi_{\mathtt{o}\to\mathtt{o}},1),[2]}$ and $g^{(\xi_{\mathtt{o}\to\mathtt{o}},2),[4]}$ are no longer confused, and the edge $([2], [4])$ is removed.

Fig. 3 shows the ACG after the refinement. In the figure, we have omitted labels (as they are important only for generating constraints), and states of $\mathcal{B}_0$ from variables. Thus, the variables g^i and x^i (where $i \in \{1, 2\}$) are actually abbreviated forms of $g^{(\xi_{\mathtt{o}\to\mathtt{o}},i),\ell}$ and $x^{(\xi_{\mathtt{o}},i),\ell'}$ respectively. We have omitted all the annotations from f, which is an abbreviation of $f^{(\xi_{\mu\alpha.\alpha\to(\mathtt{o}\to\mathtt{o})\to\mathtt{o}},1),\ell''}$. The binding function associated with the graph is as follows:

$$\rho(f^{(\xi_\kappa,1),*}) = \{F\} \quad \rho(g^{(\xi_{\mathtt{o}\to\mathtt{o}},1),*}) = \{\mathtt{b}, B\,g^{(\xi_{\mathtt{o}\to\mathtt{o}},2),*}\} \\ \rho(g^{(\xi_{\mathtt{o}\to\mathtt{o}},2),*}) = \{B\,g^{(\xi_{\mathtt{o}\to\mathtt{o}},1),*}\} \\ \rho(h^{(\xi_{\mathtt{o}\to\mathtt{o}},i),*}) = \{g^{(\xi_{\mathtt{o}\to\mathtt{o}},i),*}\} \text{ (for } i \in \{1,2\}) \\ \rho(x^{(\xi_{\mathtt{o}},1),*}) = \{x^{(\xi_{\mathtt{o}},1),*}, g^{(\xi_{\mathtt{o}\to\mathtt{o}},1),*}\mathtt{c}\} \\ \rho(x^{(\xi_{\mathtt{o}},2),*}) = \{x^{(\xi_{\mathtt{o}},2),*}, \mathtt{c}, g^{(\xi_{\mathtt{o}\to\mathtt{o}},2),*}\mathtt{c}\}$$

(Here, again we have ignored labels and merged bindings for different labels.) The graph no longer contains `fail`, so that we can conclude that $\mathbf{Tree}(\mathcal{G}_1) \in \mathcal{L}({\mathcal{A}_1}^{\perp})$. $\square$

D. Construction of the Initial Automaton

We use a HORSAT-based algorithm to construct the initial term automaton $\mathcal{B}_0$. HORSAT [8] is a model checking algorithm for (ordinary) HORS. It computes an intersection type environment Γ such that

$$\{t \mid \Gamma \vdash t : \overline{q_0}\} = \{t \mid t \longrightarrow^*_{\mathcal{G}} s \wedge s^{\perp} \notin \mathcal{L}(\mathcal{A}^{\perp})\}$$

where $\overline{q_0}$ is the initial state of the complement of $\mathcal{A}$ (thus, Γ is a finite description of the set of terms that may generate invalid trees). Γ is obtained by iteratively applying a function $\mathcal{F}$ that expands a type environment:

$$\emptyset \subseteq \mathcal{F}(\emptyset) \subseteq \mathcal{F}^2(\emptyset) \subseteq \cdots \subseteq \mathcal{F}^n(\emptyset) = \mathcal{F}^{n+1}(\emptyset) = \Gamma.$$

Then it suffices to check whether $S : \overline{q_0} \in \Gamma$. If we apply HORSAT to μHORS, we can use the same function $\mathcal{F}$, but the computation of $\mathcal{F}^n(\emptyset)$ may not converge. However, for every n, we have

$$\{t \mid t \longrightarrow^n_{\mathcal{G}} s \wedge s^{\perp} \notin \mathcal{L}(\mathcal{A}^{\perp})\} \subseteq \{t \mid \mathcal{F}^n(\emptyset) \vdash t : \overline{q_0}\} \\ \subseteq \{t \mid t \longrightarrow^*_{\mathcal{G}} s \wedge s^{\perp} \notin \mathcal{L}(\mathcal{A}^{\perp})\}.$$

Thus, for sufficiently large n, $\{t \mid \mathcal{F}^n(\emptyset) \vdash t : \overline{q_0}\}$ gives a good approximation of the set of error terms $\{t \longrightarrow^*_{\mathcal{G}} s \mid s^{\perp} \notin \mathcal{L}(\mathcal{A}^{\perp})\}$. We construct the term automaton that accepts $\{t \mid \mathcal{F}^n(\emptyset) \vdash t : \overline{q_0}\}$, and use it as the initial automaton $\mathcal{B}_0$.

Although the choice of $\mathcal{B}_0$ above is heuristic, if the input grammar $\mathcal{G}$ is an ordinary HORS and n is sufficiently large, then the refinement-loop (lines 4-15) is guaranteed to succeed immediately. According to the experimental results reported in Section V, the choice of $\mathcal{B}_0$ often helps the refinement loop terminate quickly also for μHORS.

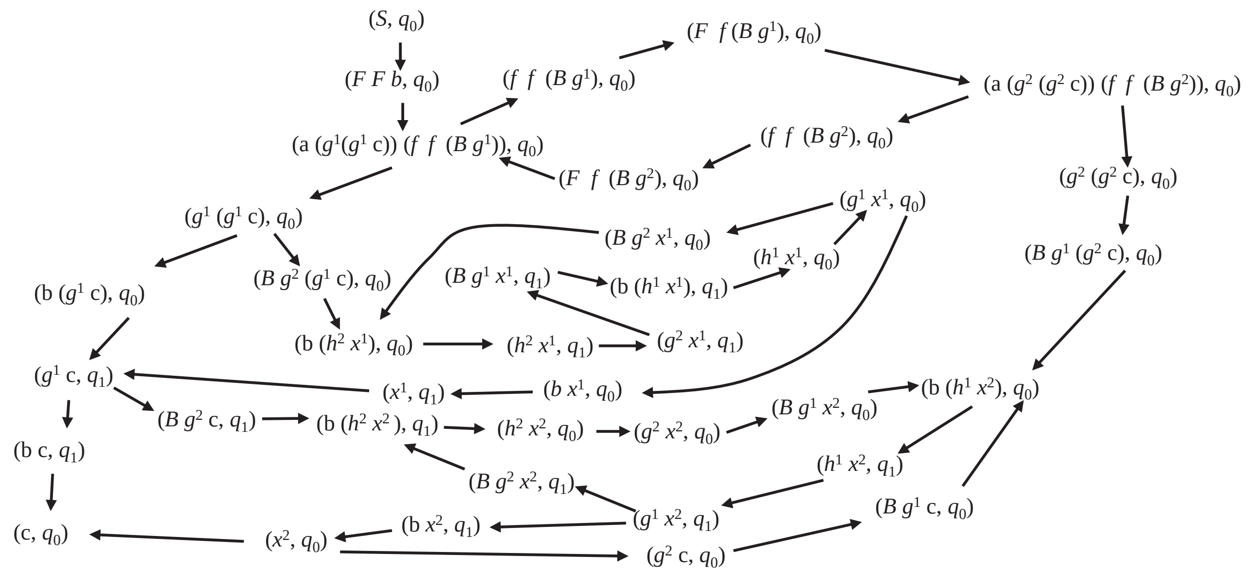


Fig. 3. Abstract configuration graph $\mathcal{C}_{\mathcal{G}_1,\mathcal{A}_1,\mathcal{B}}$.

E. Properties of the Procedure

We show that our model checking procedure is relatively complete, i.e., if a regular invariant exists, then $MC(\mathcal{G},\mathcal{A})$ terminates and reports that the property is satisfied. Note that the termination does not depend on the choice of the initial automaton (although it may affect the efficiency).

The following lemma states that if the automaton $\mathcal{B}$ used for abstraction is properly chosen, then the verification succeeds immediately.

Lemma 3.2: Suppose that I is a regular invariant for $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$, and that $\mathcal{B}$ is a term automaton that accepts I. Then $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ does not contain node `fail`.

Proof: Let ρ be the binding function associated with $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$. For each node (t,q) in $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$, we define $\mathit{Terms}_{(t,q)}$ as the least set of terms that satisfies the following conditions.

- $\mathit{Terms}_{(t,q)} \supseteq \rho^*(t)$.
  Here, $\rho^*(t)$ is the least set that satisfies the following conditions.
  $$\rho^*(x^{\xi,\ell}) \supseteq \rho^*(\rho(x^{\xi,\ell'})) \text{ if } x^{\xi,\ell'} \in \mathit{dom}(\rho)$$
  $$\rho^*(a) \supseteq \{a\} \qquad \rho^*(F) \supseteq \{F\}$$
  $$\rho^*(t_1t_2) \supseteq \{t_1't_2' \mid t_1' \in \rho^*(t_1), t_2' \in \rho^*(t_2)\}$$
- If there are edges from $(a\,t_1\,\cdots\,t_n, q)$ to (t_i, q_i) for $i \in \{1,\ldots,n\}$, then
  $$\mathit{Terms}_{(a\,t_1\,\cdots\,t_n,q)} \supseteq \{a\,t_1',\cdots\,t_n' \mid t_i' \in \mathit{Terms}_{(t_i,q_i)} \text{ for each } i\}$$
- If there is an edge from (t_1,q) to (t_2,q) where the head of t_1 is a non-terminal or a variable, then
  $$\mathit{Terms}_{(t_1,q)} \supseteq \mathit{Terms}_{(t_2,q)}.$$

Then, we have:

1) $\mathit{Terms}_{(S,q_0)} \subseteq I$.
2) If $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ contains a node `fail`, then there exists a term $t \in \mathit{Terms}_{(S,q_0)}$ such that $t^\perp \notin \mathcal{L}(\mathcal{A}^\perp)$.

The required property follows immediately from those properties. (Suppose that $\mathcal{C}_{\mathcal{G},\mathcal{A},\mathcal{B}}$ contains a node `fail`. Then there must exist a term $t \in \mathit{Terms}_{(S,q_0)} \subseteq I$ such that $t^\perp \notin \mathcal{L}(\mathcal{A}^\perp)$, but this contradicts the assumption that I is a regular invariant for $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$.) The property 2 above follows from the property that if there is a path from node (t',q) to `fail`, then there exists $t \in \mathit{Terms}_{(t',q)}$ such that $t^\perp \notin \mathcal{L}(\mathcal{A}^\perp,q)$, which follows by an easy induction on the length of the path. To show the property 1, it suffices to prove the following property:

For every node (t,q), if $t \longrightarrow^*_{\mathcal{B}} \xi$, then $\mathit{Terms}_{(t,q)} \subseteq \{s \mid \exists s_0 \in \mathcal{L}(\mathcal{B},\xi).s_0(\longrightarrow_{\mathcal{G}}\sim_{\mathcal{B}})^*s\}$.

This follows by induction on the rules for defining $\mathit{Terms}_{(t,q)}$. Suppose $t' \in \mathit{Terms}_{(t,q)}$ and $t \longrightarrow^*_{\mathcal{B}} \xi$.

- If $t' \in \rho^*(t)$, then $t' \in \mathcal{L}(\mathcal{B},\xi)$ follows from the fact that $\rho^*(x^{\xi,\ell}) \subseteq \mathcal{L}(\mathcal{B},\xi)$.
- If $t = a\,t_1\cdots t_n$ and $t' = a\,t_1',\cdots t_n'$ with $t_i' \in \mathit{Terms}_{(t_i,q_i)}$ for each i, then $a@t_1@\cdots@t_n \longrightarrow^*_{\mathcal{B}} a@\xi_1@\cdots@\xi_n \longrightarrow^*_{\mathcal{B}} \xi$; and by the induction hypothesis, there exists $s_i \in \mathcal{L}(\mathcal{B},\xi_i)$ such that $s_i(\longrightarrow_{\mathcal{G}}\sim_{\mathcal{B}})^*t_i'$. Thus, we have $s(\longrightarrow_{\mathcal{G}}\sim_{\mathcal{B}})^*t'$ for $s = a\,s_1\,\cdots\,s_n \in \mathcal{L}(\mathcal{B},\xi)$.
- If $t' \in \mathit{Terms}_{(t_2,q)}$ and there is an edge from (t,q) to (t_2,q), with the head of t being a variable, then the result follows immediately from the induction hypothesis, since $t_2 \longrightarrow^*_{\mathcal{B}} \xi$.
- If $t' \in \mathit{Terms}_{(t_2,q)}$ and there is an edge from (t,q) to (t_2,q), with the head of t being a non-terminal, then $t = F\,s_1\,\cdots\,s_n$ and $t_2 = [x_1^{\xi_1,\ell_1}/x_1,\ldots,x_n^{\xi_n,\ell_n}/x_n]u$ with $F\,x_1\,\cdots\,x_n \to u \in \mathcal{R}_{\mathcal{G}}$. Pick $s_i' \in \mathcal{L}(\mathcal{B},\xi_i)$ for each i. Then by the induction hypothesis, we have $[s_1'/x_1,\ldots,s_n'/x_n]u \sim_{\mathcal{B}} (\longrightarrow_{\mathcal{G}}\sim_{\mathcal{B}})^*t'$. Thus, $\mathcal{L}(\mathcal{B},\xi) \ni F\,s_1'\,\cdots\,s_n'(\longrightarrow_{\mathcal{G}}\sim_{\mathcal{B}})^*t'$ as required.

■

Theorem 3.3: If there is a regular invariant for $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$, then $MC(\mathcal{G}, \mathcal{A})$ eventually terminates and reports that the property is satisfied.

Proof: Let $\mathcal{B}'$ be a term automaton that accepts the regular invariant, and $\mathcal{B}_0$ be the initial automaton chosen on line 2. Then the product automaton $\mathcal{B}$ of $\mathcal{B}_0$ and $\mathcal{B}'$ also accepts the regular invariant. Let k be the number of states of $\mathcal{B}'$. Then there is a k-refinement $\mathcal{B}_1$ of $\mathcal{B}_0$ that is equivalent to $\mathcal{B}$. Since the number of k'-refinement is finite for each $k' \leq k$, and each refinement loop picks a different automaton $\mathcal{B}$ (to see the latter, note that the constraint $\bigvee_{(\ell,\ell')\in CE} \delta_{\mathcal{B}}(\rho^{+}(\ell)) \neq \delta_{\mathcal{B}}(\rho^{+}(\ell'))$ added at each step is not satisfied by the current automaton but satisfied by the next refinement automaton), the refinement loop eventually terminates or picks $\mathcal{B}_1$. In the latter case, by Lemma 3.2, the refinement loop terminates and reports that the property is satisfied. ■

IV. On Relative Completeness

In this section, we show the equivalence of the two assumptions for relative completeness: the existence of a regular invariant (used in this paper) and the typability in Kobayashi and Igarashi's recursive intersection type system (used in [7]).

We first recall their recursive intersection type system. The set of recursive intersection types is defined by the syntax:

$$\begin{aligned}\theta &::= \textstyle\bigwedge \Theta_1 \rightarrow \cdots \rightarrow \bigwedge \Theta_m \rightarrow q \mid \alpha \mid \mu\alpha.\theta \\ \Theta &::= \{\theta_1, \ldots, \theta_k\}\end{aligned}$$

A type environment is a set of bindings of the form $x : \theta$, which may contain more than one binding for each variable. The typing rules are given as follows:

$$\frac{\theta \leq \theta'}{\Gamma, x : \theta \vdash_{\mathcal{A}} x : \theta'}$$

$$\frac{(q, a, q_1 \cdots q_k) \in \delta_{\mathcal{A}} \qquad q_1 \rightarrow \cdots \rightarrow q_k \rightarrow q \leq \theta}{\Gamma \vdash_{\mathcal{A}} a : \theta}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{A}} t_1 : \bigwedge\{\theta_1, \ldots, \theta_k\} \rightarrow \theta \\ \Gamma \vdash_{\mathcal{A}} t_2 : \theta'_i \text{ and } \theta'_i \leq \theta_i \text{ (for every } i \in \{1, \ldots, k\})\end{array}}{\Gamma \vdash_{\mathcal{A}} t_1 t_2 : \theta}$$

Here, we omit the definition of the subtyping relation $\theta \leq \theta'$ and refer the reader to [7]. We write $\Gamma \vdash_{\mathcal{A}} \mathcal{G}$ if (i) $S : q_0 \in \Gamma$, and (ii) for each $F\, x_1 \cdots x_n \rightarrow t \in \mathcal{R}_{\mathcal{G}}$ and $F : \theta \in \Gamma$, θ is of the form $\bigwedge \Theta_1 \rightarrow \cdots \rightarrow \bigwedge \Theta_n \rightarrow q$ and $\Gamma \cup \{x_i : \theta \mid \theta \in \Theta_i, i \in \{1, \ldots, n\}\} \vdash t : q$. The type system is sound (but not complete) with respect to the model checking problem, i.e., if $\Gamma \vdash_{\mathcal{A}} \mathcal{G}$ then $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$.

The main result in [7] was a model checking procedure that is relatively complete with respect to the typability: if there exists Γ such that $\Gamma \vdash_{\mathcal{A}} \mathcal{G}$, then their procedure eventually finds one.

We can show that the two assumptions for relative completeness are equivalent. (Understanding the detail of the proof requires familiarity with [7].)

Theorem 4.1: Let $\mathcal{G}$ be a μHORS and $\mathcal{A}$ be a (topdown-deterministic) tree automaton. Then, there is a regular invariant for $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$ if and only if there exists a type environment Γ such that $\Gamma \vdash_{\mathcal{A}} \mathcal{G}$.

Proof Sketch: The "if" direction follows immediately from the fact that $\{t \vdash \Gamma \vdash_{\mathcal{A}} t : q_0\}$ is a regular invariant. The set being closed under reduction follows immediately from the standard subject reduction property. The proof that the set is regular is the same as the case without recursive types [14], [8]: one can construct an alternating tree automaton that accepts the set, having states corresponding to the intersection types occurring in the derivation of $\Gamma \vdash_{\mathcal{A}} \mathcal{G}$.

The "only if" direction follows from the proof of relative completeness in [7]. Let $\mathcal{B}$ be a term automaton that accepts the regular invariant. By choosing $\sim_{\mathcal{B}}$ as the term equivalence relation of [7], we can obtain a fair reduction sequence $(\mathcal{X}_0,\mathcal{U}_0) \longrightarrow_{\sim_{\mathcal{B}}} (\mathcal{X}_1,\mathcal{U}_1) \longrightarrow_{\sim_{\mathcal{B}}} (\mathcal{X}_2,\mathcal{U}_2) \longrightarrow_{\sim_{\mathcal{B}}} \cdots$ of [7] that does not contain `fail`, from which we can extract Γ ($\Gamma_{\mathcal{X},\mathcal{U},\sim_{\mathcal{B}}}$ in [7]) such that $\Gamma \vdash_{\mathcal{A}} \mathcal{G}$. □

V. Implementation and Experiments

We have implemented a prototype model checker called MuHorSar based on the new procedure described in Section III, and the tool is written in OCaml. As for the underlying SMT solver, we have used Z3 4.3.3 (http://z3.codeplex.com/).

We have evaluated the tools on benchmarks from two categories of applications, including verification problems of FJ (Featherweight Java) programs detailed in Section V-A and that of multi-threaded programs with recursion detailed in V-B. For the former, we reused the reduction and translator from FJ to μHORS presented in [7]. To show the effectiveness of the new algorithm, we enlarged the benchmarks with programs implementing container classes like stack and queue, and applications based on them. As for the latter, we inherited the approach of [7] for verifying concurrent programs that explicitly writes a thread-scheduler in μHORS. Notably, we extended the target programs with boolean variables (i.e., all variables and function parameters have boolean type), if-statement and while-loop that are guarded by (negated) boolean variables and have deterministic choices for their conditions. By such an extension, we are able to verify interesting problems like mutual exclusion and deadlock. We have also implemented a translator in Java from multi-threaded boolean programs with recursion (also written in Java syntax) to μHORS. To facilitate the translation, we first translated the target program to Jimple (a typed 3-address representation of Java suitable for analysis) by Soot (http://sable.github.io/soot/). Our benchmarks are available at http://www-kb.is.s.u-tokyo.ac.jp/~li-xin/muhorsar.

The preliminary experimental results for comparing the verification time taken by MuHorSar and RTRecS are given in Table I-III. For all the tables, the column "Bench" lists the name of studied benchmarks. The columns "#G" and "#A" show the number of rules of μHORS for each benchmark and the size of the property automaton in question, respectively. The column "R" gives the answer whether the property is satisfied (Y) or violated (N). The last two columns "MuHorSar"

TABLE I
RESULTS FOR VERIFYING BENCHMARKS FROM [7]

Bench	#G	#A	R	MUHORSAR Horsat (#Ar)	MUHORSAR Sort (#Ar)	RTRECS
$\mathcal{G}_1$	2	2	Y	0.006	0.005	0.009
$\mathcal{G}_2$	3	2	Y	0.004	0.004	0.010
Thread	9	5	Y	0.013	0.040 (1)	0.181
Pred	15	1	Y	0.008	0.007	0.010
Ski1	22	1	N	0.008	0.007	0.005
Ski2	25	1	Y	0.008	0.007	0.010
L-append	30	1	Y	0.013	0.012	0.012
L-map	182	1	Y	0.561	0.563	0.189
L-app-map	212	1	Y	0.840	0.831	0.279
L-even	87	1	Y	0.077	0.059	0.021
L-filter	122	1	Y	1.454 (6)	3.814 (33)	0.429
L-risers	122	1	Y	1.450 (6)	4.219 (33)	0.431
Twofiles	21	5	Y	0.027	0.187 (6)	4.390

and "RTRECS" give the time taken by each tool for verification, respectively. The column "MUHORSAR" has two subcolumns "HORSAT (#Ar)" and "Sort (#Ar)" that shows the time with the initial automaton being constructed by HORSAT (as explained in Section III-D) and the type information (as in in Example 3.1), respectively, where "#Ar" records the number of refinement steps taken by verification if any. The timeout is set to be 5 minutes. We make the best of each tool for verification by tuning their parameters (e.g., the refinement steps for MUHORSAR or the reduction steps for RTRECS) when necessary. Time is given in seconds or "-" for timeout. All benchmarks were run on a machine having a Mac OS X v.10.9.2, 1.7 GHz Intel Core i7 processor and 8GB RAM.

Please note that the current implementation of MUHORSAR is very naive; in particular, the tool currently uses the text interface of Z3 for the sake of simplicity, and forks a Z3 process through OS system call in each abstraction-refinement loop. Thus, in the experimental results reported below, much of the time is spent for calling Z3 when the number of refinement loops (#Ar) is large. The performance would be substantially enhanced by using the library interface of Z3 for solving the constraints incrementally.

For all the negative benchmarks in Tables II and III, we have manually added a dummy branch that has a very long reduction sequence. More precisely, we have added the following rule:

$$S' \to \texttt{br}\ \mathit{Dummy}\ S$$

where S is the start symbol of the original grammar, S' is the new start symbol, br is a terminal symbol for encoding the non-deterministic choice, and the non-terminal *Dummy* has a very long reduction sequence that only generates a valid tree. This is for emphasizing the advantage of the new mechanism in finding a counterexample. Since the size of each benchmark is not large, the naive exhaustive search as implemented in RTRECS actually works for the original μHORS. The twist explained above makes the naive search of RTRECS fail, while it has little impact on MUHORSAR.

A. Verifying Functional Objects

Table I summarizes the experimental results on benchmarks originated from [7]. Most of benchmarks as given in the second row are for FJ programs, except for "$\mathcal{G}_1$", "$\mathcal{G}_2$" and "Thread". We refer to [7] for details of those benchmarks. For this family of benchmarks, both MUHORSAR and RTRECS successfully verified all programs. For the top six benchmarks except for "Thread", RTRECS performs equally well (slightly worse) as MUHORSAR . For benchmarks having their names prefixed by "L-", RTRECS slightly outperforms MUHORSAR. As the size of the property automaton increases for verifying "Thread" and "Twofiles", MUHORSAR outperforms RTRECS by orders of magnitude. As for the two choices of the initial automaton, MUHORSAR performs equally well for verifying most benchmarks, except for the last three examples which MUHORSAR obviously found more difficult to verify (with more time and refinement steps) using the type-based construction.

TABLE II
RESULTS FOR VERIFYING FJ PROGRAMS

Bench	#G	#A	R	MUHORSAR Horsat (#Ar)	MUHORSAR Sort (#Ar)	RTRECS
stack	33	1	Y	0.040	0.075 (2)	0.207
		3		0.039	0.071 (2)	3.435
		5		0.044	0.066 (2)	23.292
stack-br	39	1	Y	0.396 (13)	0.672 (24)	-
		3		0.403 (13)	0.671 (24)	-
		5		0.397 (13)	0.777 (28)	-
queue	56	1	Y	0.169	0.165	0.143
		3		0.173	0.165	2.140
		5		0.164	0.167	12.633
queue-br	61	1	Y	0.249 (2)	0.324 (6)	-
		3		0.249 (2)	0.307 (5)	-
		5		0.249 (2)	0.373 (8)	-
queue-pc	104	1	Y	1.160	1.133	0.211
		3		1.218	1.130	0.642
		5		1.137	1.153	1.648
2stack	43	1	Y	0.151	0.144	0.052
		5		0.145	0.137	0.285
2stack-e	52	1	N	0.105	0.098	-
		5		0.105	0.101	-
2stack-br	44	1	Y	0.261	0.325 (2)	0.115
		5		0.253	0.320 (2)	6.459
2stack-pc	88	1	Y	4.202	4.204	0.498
		5		4.176	4.309	1.262
		7		4.182	4.179	2.132
2stack-pc-e	97	1	N	0.776	0.761	-
		5		0.768	0.764	-
nat	35	1	Y	17.810 (147)	-	0.288

Table II summarizes the experimental results on benchmarks of FJ programs that implement stacks and queues, and applications based on them. The example "stack" pushes a sequence of string and integer objects onto the stack, followed by a sequence of popping, and then reads the topmost stack symbol and checks whether it is a string or an integer object. We verified the last-in-first-out property of stacks by verifying the program does not fail (i.e., it never attempts to get an integer object when the topmost is a string object, and vice versa). Similarly, we verified the first-in-first-out property of queues by verifying the program does not fail. The examples "stack-br" and "queue-br" are variants of "stack" and "queue", respectively. The example "queue-pc" implements a producer and a consumer that alternatively put and take an item from a queue. We verified that the program does not fail (i.e., never dequeue an empty queue). To confirm the bad behaviour of RTRECS as the size of the property automaton increases, we gradually refined the automaton by introducing more states.

The benchmarks having their name prefixed by "2stack-" implement a pseudo-queue using two stacks, say *in-stack* and *out-stack*: for enqueue, it pushes onto *in-stack*, and for dequeue, it non-deterministically pops from *out-stack* or pops an element from *in-stack* and pushes it onto *out-stack*. The example "2stack-br" simply puts in a non-deterministic if-statement with two different sequences of enqueue and dequeue operations. The example "2stack-pc" implements a producer and a consumer that alternatively put and take an item from a pseudo-queue. The example "2stack-e" attempts to dequeue a pseudo-queue having an empty *out-stack*. For this category of benchmarks, we verified that the program does not fail (i.e., never attempt to dequeue a pseudo-queue having an empty *out-stack*), and that the program satisfies the temporal property that *dequeue* must be followed by *pop* and *enqueue* must be followed by *push*, respectively. The last example "nat" implements natural numbers and we verified that "$3 \times n$ modulo 3 = 0" by verifying that the program does not fail.

The experimental results in Table II proves our claim that RTRECS does not scale well as the size of the property automaton increases. When the size of the property automaton is relatively larger, MUHORSAR often outperforms RTRECS in orders of magnitude. It is also not effective for RTRECS to find counter-examples by a naive search of the state space. For this family of benchmarks, MUHORSAR successfully verified all benchmarks although it has some difficulty in verifying "nat", whereas RTRECS couldn't terminate for many benchmarks.

B. Verifying Multi-threaded Boolean Programs with Recursion

Table III shows experimental results for verifying a family of multi-threaded boolean programs with recursion. The reachability problem of multi-threaded programs with recursion is known to be undecidable. For simplicity, we assume that there are two threads asynchronously running for all the benchmarks. As [7], we modeled concurrency by explicitly writing a thread-scheduler in μHORS, and represent the thread of control as a continuation. We modeled the interleaving semantics of multi-threaded programs and the thread of control is passed to each thread non-deterministically. We improved [7] by further considering boolean programs, and modeled boolean variables and parameters in a store-passing style. Thanks to recursive types, we are able to model recursive programs in μHORS. When there is no recursion in the studied examples, we artificially added two mutually recursive functions at a proper place in the program and allow the functions to terminate non-deterministically.

We studied properties of mutual exclusion, deadlock, and assertion checking (reduced to reachability checking), and classified benchmarks into two rows by having positive or negative answers. To study concurrent properties, we implemented in the target programs the basic simplified usage of synchronization mechanisms *monitor* (mutex with blocking *wait()*, and *notify()* functions), and *binary semaphore*. To translate the target programs to μHORS, (i) when one thread has an exclusive access to a monitor or atomic operations of a semaphore, we do not allow the interleaved execution of another thread, whereas (ii) the thread of control is deterministically (resp. non-deterministically) passed to another thread right after one thread calls *wait()* (resp. *notify()*).

TABLE III
RESULTS FOR VERIFYING MULTI-THREADED BOOLEAN PROGRAMS WITH RECURSION

Bench	#G	#A	R	MUHORSAR		RTRECS
				Horsat (#Ar)	Sort (#Ar)	
locks-e	103	5	N	0.160	0.625	-
dining-e	135	5	N	2.857	-	-
dining-sp-e	193	5	N	10.997	-	-
bluetooth	129	1	N	2.300	-	-
bluetooth-v	158	1	N	272.626	-	-
locks	95	5	Y	0.779	-	-
plotter	88	4	Y	0.195	0.240 (3)	1.189
peterson	74	2	Y	3.331 (2)	-	-
peterson-d	80	9	Y	-	-	-
dekker	94	2	Y	-	-	-
pc-monitor	71	5	Y	0.338	-	-
pc-sp	111	5	Y	2.250	-	-
dining-sp	303	5	Y	-	-	-

The benchmark "locks-e" implements the notorious example that one thread acquires locks l_1 and l_2 in order, and another thread acquires locks l_2 and l_1 in an opposite order. We verified that it gives rise to a deadlock. The programs "dining-e" and "dining-sp-e" implement the dining philosophers problem, respectively, and both are wrong implementations and may cause a deadlock. The property automaton for verifying them specifies that, there is at most one philosopher waiting for the chopstick to his right or left. The program "bluetooth" is an example taken from [15], and is a simplified model of Bluetooth drivers. A driver has four global variables that are shared among threads, and a thread could modify the value of global variables and either stops the driver or performs I/O in the driver. We verified that the program does not fail (i.e., never violates assertions in the program). The example "bluetooth-v" is a variant of "bluetooth" that uses locks for mutual exclusion, although it would still reach an error state.

The benchmark "locks" is a correct program for using nested locks and is deadlock-free. The program "plotter" is an example studied in the literature of pushdown systems [16]. We verified that it satisfies some temporal property. The benchmarks "peterson" and "dekker" implement the well-known Peterson's algorithm and Dekker's algorithm for mutual exclusion, respectively. The property automaton for them specifies that at most one thread can reside in the critical section at the same time. For the last three benchmarks in the table, we attempted to verify that they are deadlock-free. The program "pc-monitor" and "pc-sp" implements the producer-consumer problem using monitor and binary semaphore, respectively. The program "dining-sp" implements Tannenbaum's correct solution to the dining philosophers problem using binary semaphore. The property automaton for those programs specifies that at most one thread is waiting. For "peterson-d", we attempted to verify that Peterson's algorithm is deadlock free. Because two threads could interleave arbitrarily without synchronization and the algorithm uses non-blocked busy waiting, the property automaton for verification is slightly more complex and specifies that, at most one thread is busy waiting after a finite sequences of alternative (positively consecutive) waiting of two threads.

For this family of benchmarks, MUHORSAR verified many benchmarks while RTRECS failed verifying most of them within the given timeout, including all benchmarks having negative answers. The choice of the initial automaton also matters a lot to verification, and MUHORSAR does not scale well using the type-based construction.

VI. Related Work

As already discussed, most closely related to the present work is Kobayashi and Igarashi's previous work on μHORS model checking. Our new procedure is quite different from theirs, and as confirmed by the experiments, ours is often more efficient. Another contribution is that we have replaced their assumption for relative completeness (which is the typability in their own recursive intersection type system) with an (arguably) more familiar one (the existence of an inductive invariant that is a regular language).

Our model checking procedure has been inspired by the recent model checking algorithms PREFACE [9] and HORSAT [8] for ordinary HORS. In particular, the idea of our abstract configuration graph (based on abstract binding) has been borrowed from PREFACE. The main difference is in abstraction; PREFACE collects two kinds of intersection types (acceptance types and rejection types) and use them for abstracting terms, whereas our procedure uses an automaton for abstracting terms. The completeness of both PREFACE and HORSAT relies on the fact that there can be finitely many intersection types, which are not the case in the setting of μHORS. That is why we had to use automata to guarantee relative completeness. As a result, the refinement procedure is completely different.

A few other extensions of HORS have been introduced and verification methods have been considered. Kobayashi et al. [17] introduced an extension of HORS called higher-order multi-tree transducers (HMTT), and proposed a verification method based on automata-based abstraction. The original work did not employ abstraction refinement, but it has recently been extended to employ counterexample-guided abstraction refinement [18]. Their method is not directly applicable or comparable due to a number of differences; the underlying computation model of μHORS and HMTT (in particular, HMTT is simply-typed, while μHORS has recursive types); automata are used to abstract tree data in the HMTT verification procedure, whereas they are used to abstract terms in our procedure. Ong and Ramsay [5] also introduced an extension of HORS with pattern matching, called PMRS, and proposed a verification procedure based on counterexample-guided abstraction refinement (CEGAR). Their abstraction is based on finite patterns, which is weaker than our automata-based abstraction (in the sense that finite patterns can be expressed by automata, but not vice versa). They do not guarantee relative completeness in the sense of ours. (They guarantee that a counterexample is eventually found if there exists one, but do not guarantee the success of verification.)

Another line of related work is tree automata completion for term rewriting systems (TRS) [19], [20], [21], [22]. Tree automata completion computes a tree automaton that represents an over-approximation of the set of reachable terms by a TRS. Therefore, one can check that no error term is reachable, by first performing tree automata completion, and then checking that the automaton accepts no error term. Since a μHORS can be easily translated to a TRS (by representing a function application $t_1 t_2$ as $@(t_1, t_2)$), one can use a tree automata completion procedure as a sound (but incomplete) μHORS model checking procedure. Boichut et al. [23] has proposed a counterexample-guided abstraction refinement for tree automata completion. To our knowledge, however, no tree automata completion procedure has been proposed that satisfies a similar relative completeness condition.

Our analysis is also related to flow analysis for functional programs, although we are not aware of any method that guarantees relative completeness in the sense of ours. Jones and Anderson [24] proposed a method for over-approximating the set of reachable terms by a tree grammar. This seems essentially equivalent to a special case of our abstract configuration graph where the term automaton is collapsed to an automaton with just one state (so that all the terms are abstracted to the same element). In k-CFA [25], variable bindings are distinguished based on calling contexts, while we (like Preface [9]) distinguish them based on the abstraction of the values of variables. It is actually easy to extend our procedure to take calling contexts into account.

VII. Conclusion

We have proposed a new procedure for μHORS model checking and proved that it is relatively complete with respect to the existence of a regular invariant. We have implemented the new procedure and confirmed that it often outperforms the previous procedure for μHORS model checking.

A limitation of μHORS model checking is the weak relative completeness guarantee that invariants must be regular. Consider, for example, the following μHORS (extended with natural numbers and conditionals, which can be encoded in μHORS):

$$\begin{array}{l} S \to F\,0\,0 \\ F\,m\,n \to \mathtt{a}\,(F\,(m+1)\,(n+1))\,(G\,m\,n) \\ G\,m\,n \to \mathtt{if}\ m=n\ \mathbf{then}\ \mathtt{end}\ \mathbf{else}\ \mathtt{fail}. \end{array}$$

Suppose that the property automaton accepts the set of trees that do not contain fail. Then, the above grammar satisfies the property, but there is no regular invariant, since any inductive invariant contains $F\,m\,n$ if and only if $m = n$, which is not regular (assuming that a natural number n is represented as $\mathit{Succ}^n\,\mathit{Zero}$). Thus, our μHORS model checking procedure (as well as the previous one [7]) would fail. To address this issue, we plan to combine μHORS model checking with predicate abstraction, just like the ordinary HORS model checking has been combined with predicate abstraction [4].

Acknowledgment

This work is partially supported by Kakenhi 23220001 and 25730039. The authors would like to thank Atsushi Igarashi and anonymous referees for useful comments.

References

[1] T. Knapik, D. Niwinski, and P. Urzyczyn, “Higher-order pushdown trees are easy,” in *Proceedings of FoSSaCS 2002*, ser. Lecture Notes in Computer Science, vol. 2303. Springer, 2002, pp. 205–222.

[2] C.-H. L. Ong, “On model-checking trees generated by higher-order recursion schemes,” in *Proceedings of IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2006, pp. 81–90.

[3] N. Kobayashi, “Model checking higher-order programs,” *Journal of the ACM*, vol. 60, no. 3, 2013.

[4] N. Kobayashi, R. Sato, and H. Unno, “Predicate abstraction and CEGAR for higher-order model checking,” in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2011, pp. 222–233.

[5] C.-H. L. Ong and S. Ramsay, “Verifying higher-order programs with pattern-matching algebraic data types,” in *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2011, pp. 587–598.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[7] N. Kobayashi and A. Igarashi, “Model checking higher-order programs with recursive types,” in *Proceedings of ESOP 2013*, ser. Lecture Notes in Computer Science, vol. 7792. Springer, 2013.

[8] C. H. Broadbent and N. Kobayashi, “Saturation-based model checking of higher-order recursion schemes,” in *Proceedings of CSL 2013*, ser. LIPIcs, vol. 23, 2013, pp. 129–148.

[9] S. Ramsay, R. Neatherway, and C.-H. L. Ong, “A type-directed abstraction refinement approach to higher-order model checking,” in *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2014, pp. 61–72.

[10] R. M. Amadio and L. Cardelli, “Subtyping recursive types,” *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 4, pp. 575–631, September 1993.

[11] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, “Tree automata techniques and applications,” Available on: http://www.grappa.univ-lille3.fr/tata, 2007, release October, 12th 2007.

[12] G. D. Plotkin, “Call-by-name, call-by-value and the lambda-calculus,” *Theor. Comput. Sci.*, vol. 1, no. 2, pp. 125–159, 1975.

[13] N. Kobayashi, “Model-checking higher-order functions,” in *Proceedings of PPDP 2009*. ACM Press, 2009, pp. 25–36.

[14] J. Rehof and P. Urzyczyn, “Finite combinatory logic with intersection types,” in *Proceedings of TLCA 2011*, ser. Lecture Notes in Computer Science, vol. 6690. Springer, 2011, pp. 169–183.

[15] S. Qadeer and D. Wu, “Kiss: Keep it simple and sequential,” in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2004, pp. 14–24.

[16] S. Schwoon, “Model-checking pushdown systems,” Ph.D. dissertation, Technische Universität München, 2002.

[17] N. Kobayashi, N. Tabuchi, and H. Unno, “Higher-order multi-parameter tree transducers and recursion schemes for program verification,” in *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2010, pp. 495–508.

[18] Y. Matsumoto, N. Kobayashi, and H. Unno, “Counterexample finding and abstraction refinement for automated verification of higher-order transducers,” *Computer Software*, 2015, in press (in Japanese).

[19] G. Feuillade, T. Genet, and V. V. T. Tong, “Reachability analysis over term rewriting systems,” *J. Autom. Reasoning*, vol. 33, no. 3-4, pp. 341–383, 2004.

[20] T. Takai, “A verification technique using term rewriting systems and abstract interpretation,” in *Proceedings of RTA 2004*, ser. Lecture Notes in Computer Science, vol. 3091. Springer, 2004, pp. 119–133.

[21] T. Genet and V. Rusu, “Equational approximations for tree automata completion,” *J. Symb. Comput.*, vol. 45, no. 5, pp. 574–597, 2010.

[22] T. Genet, “Towards static analysis of functional programs using tree automata completion,” in *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014*, ser. Lecture Notes in Computer Science, vol. 8663. Springer, 2014, pp. 147–161.

[23] Y. Boichut, B. Boyer, T. Genet, and A. Legay, “Equational abstraction refinement for certified tree regular model checking,” in *Proceedings of ICFEM 2012*, ser. Lecture Notes in Computer Science, vol. 7635. Springer, 2012, pp. 299–315.

[24] N. D. Jones and N. Andersen, “Flow analysis of lazy higher-order functional programs,” *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 120–136, 2007.

[25] O. Shivers, “Control-flow analysis of higher-order languages,” Ph.D. dissertation, Carnegie-Mellon University, May 1991.