Model-Checking Higher-Order Programs with Recursive Types

Naoki Kobayashi[1] and Atsushi Igarashi[2]

[1] The University of Tokyo
[2] Kyoto Univeristy

Abstract. Model checking of higher-order recursion schemes (HORS, for short) has been recently studied as a new promising technique for automated verification of higher-order programs. The previous HORS model checking could however deal with only *simply-typed* programs, so that its application was limited to functional programs. To deal with a broader range of programs such as object-oriented programs and multi-threaded programs, we extend HORS model checking to check properties of programs with *recursive* types. Although the extended model checking problem is undecidable, we develop a sound model-checking algorithm that is relatively complete with respect to a recursive intersection type system and prove its correctness. Preliminary results on the implementation and applications to verification of object-oriented programs and multi-threaded programs are also reported.

1 Introduction

The model checking of higher-order recursion schemes (HORS for short) [15] has been recently studied as a new technique for automated verification of higher-order functional programs [9, 14, 16, 13]. HORS is essentially a simply-typed higher-order functional program with recursion for generating (possibly infinite) trees, and the goal of HORS model checking is to decide whether the tree generated by a given HORS satisfies a given property. The idea of applying the HORS model checking is to transform a given functional program M to a HORS $\mathcal{G}$ that generates a tree describing possible outputs or event sequences of the program [9]; verification of the program is then reduced to HORS model checking, to decide whether the tree generated by $\mathcal{G}$ represents valid outputs or event sequences. Based on this idea, various verification problems for functional programs have been reduced to it [9, 14, 16]. By combining it with predicate abstraction, a software model checker for functional programs can be constructed [16, 13].

The above approach to automated verification of functional programs, however, cannot be smoothly extended to support other important programming language features, such as objects and concurrency. Object-oriented programs often use (mutually) recursive interfaces, which cannot be naturally modeled by HORS (which are *simply-typed* functional programs). In fact, even Featherweight Java (FJ) [5] (with only objects as primitive data) is Turing complete [22]. As for concurrency, the model checking of concurrent pushdown systems [20] is undecidable. These imply that there cannot be a sound and complete reduction from verification problems for object-oriented or recursive concurrent programs to HORS model checking. These situations are in sharp contrast to the case for functional programs, for which we have a sound and complete reduction to HORS model checking, as long as the programs use only finite base types (such as booleans, but not unbounded integers) [9].

The present paper aims to overcome the above limitations by introducing an extension of HORS model checking, where models, i.e., higher-order recursion schemes, are extended with recursive types. The extended higher-order recursion schemes, called *μHORS*, are essentially the simply-typed λ-calculus extended with tree constructors, (term-level) recursion, and recursive types, which is Turing complete. The model checking of μHORS (μHORS model checking for short) is undecidable, but we can develop a sound (but incomplete) model checking procedure. The procedure uses the result that HORS model checking can be reduced to a type checking problem in an intersection type system [9, 11, 24], and solves the type checking problem. Although the procedure is incomplete (as μHORS model checking is undecidable) and may not terminate, it is relatively complete with respect to a certain recursive intersection type system: any program that is typable in the type system is eventually proved correct. The procedure incorporates a novel reduction of the intersection type checking to SAT solving, which may be of independent interest and applicable to ordinary HORS checking.

Being armed with μHORS model checking, we can construct a fully automated verification tool (or so called a "software model checker") for various programming languages. Given a program, we first apply a kind of program transformation to get a μHORS that generates a tree describing all the possible program behaviors of interest, and then use μHORS model checking to check that the tree describes only valid behaviors. As a proof of concept, we have implemented a prototype of the μHORS model checker and a translator from Featherweight Java (FJ) programs [5] to μHORS. Preliminary experiments show that we can indeed use the μHORS model checker to verify small but non-trivial object-oriented programs.

For the space restriction, we omit some examples and proofs, which are found in an extended version [10].

2 Preliminaries

This section introduces μHORS, defines model checking problems for them, and reduces it to a type-checking problem in a recursive intersection type system.

2.1 Recursive Intersection Types

Before introducing μHORS model checking, we first formalize recursive intersection types. We fix a finite set Q of base types below, and use the meta-variable q for its elements. We use the meta-variable α for type variables.

Definition 1. *A* (recursive intersection) type *is a pair (E, α), where E is a finite set of equations of the form $\alpha_i = \sigma_1 \to \cdots \to \sigma_m \to q$, and σ is of the form $\bigwedge\{\alpha_1, \ldots, \alpha_k\}$. Here m and k may be 0. We use the meta-variable τ for recursive intersection types. We write $\mathbf{Tv}(\tau)$ for the set of type variables occurring in τ. A recursive intersection type $\tau = (E, \alpha)$ is* closed *if, for every $\alpha \in \mathbf{Tv}(\tau)$, $(\alpha = \theta) \in E$ for some θ. When $(\alpha = \theta) \in E$, we write $E(\alpha)$ for θ.*

We identify types up to renaming of type variables. For example, $(\{\alpha = q\}, \alpha)$ is the same as $(\{\beta = q\}, \beta)$. Thus, for two closed types τ_0 and τ_1, we always assume that $\mathbf{Tv}(\tau_0) \cap \mathbf{Tv}(\tau_1) = \emptyset$. We often write $\alpha_1 \wedge \cdots \wedge \alpha_k$ or $\bigwedge_{i \in \{1,\ldots,k\}} \alpha_i$ for $\bigwedge\{\alpha_1, \ldots, \alpha_k\}$ and write $\top$ for $\bigwedge \emptyset$. Intuitively, (E, α) denotes the (recursive) type α that satisfies the equations in E. For example, $(\{\alpha = \alpha \to q\}, \alpha)$ represents the recursive type $\mu\alpha.(\alpha \to q)$ in the usual notation. We often use this term notation for recursive intersection types. By abuse of notation, when $E(\alpha) = \bigwedge_{i \in I_1} \alpha_i \to \cdots \to \bigwedge_{i \in I_k} \alpha_i \to q$, we write $\bigwedge_{i \in I_1} (E, \alpha_i) \to \cdots \to \bigwedge_{i \in I_k} (E, \alpha_i) \to q$ for (E, α). For example, when $E = \{\alpha_0 = \alpha_1 \to q_0, \alpha_1 = q_1\}$, (E, α_0) is also written as $(E, \alpha_1) \to q_0$ or $q_1 \to q_0$. The type $\sigma_1 \to \cdots \to \sigma_m \to q$ describes functions that take m arguments of types $\sigma_1, \ldots, \sigma_m$, and return a value of type q. The type $\alpha_1 \wedge \cdots \wedge \alpha_k$ describes values that have all of the types $\alpha_1, \ldots, \alpha_k$. For example, if $Q = \{q_1, q_2\}$, the identity function on base values ($\lambda x.x$ in the λ-calculus notation) would have types $(q_1 \to q_1) \wedge (q_2 \to q_2)$.

We define the subtyping relation $\tau_0 \leq \tau_1$, which intuitively means, as usual, that any value of type τ_0 can be used as a value of type τ_1.

Definition 2 (subtyping). *Let $\tau = (E'', \alpha)$ and $\tau' = (E', \alpha')$ be closed types, and let $E = E'' \cup E'$. The type τ is a* subtype *of τ', written $\tau \leq \tau'$, if there exists a binary relation $\mathcal{R}$ on $\mathbf{Tv}(\tau) \cup \mathbf{Tv}(\tau')$ such that (i) $(\alpha, \alpha') \in \mathcal{R}$ and (ii) for every $(\alpha_0, \alpha'_0) \in \mathcal{R}$, there exist $\sigma_1, \ldots, \sigma_m, \sigma'_1, \ldots, \sigma'_m, q$ such that $E(\alpha_0) = \sigma_1 \to \cdots \to \sigma_m \to q$ and $E(\alpha'_0) = \sigma'_1 \to \cdots \to \sigma'_m \to q$, with $(\sigma'_1, \sigma_1), \ldots, (\sigma'_m, \sigma_m) \in \mathcal{R}^{\wedge}$. Here, $\mathcal{R}^{\wedge}$ is:*
$\{(\bigwedge\{\alpha'_1, \ldots, \alpha'_{k'}\}, \bigwedge\{\alpha_1, \ldots, \alpha_k\}) \mid \forall i \in \{1, \ldots, k\}.\exists j \in \{1, \ldots, k'\}.\alpha'_j \mathcal{R} \alpha_i\}$.
We write $\tau \cong \tau'$ if $\tau \leq \tau'$ and $\tau' \leq \tau$.

Example 1. Let $\tau_0 = (\{\alpha_0 = \alpha_0 \to q\}, \alpha_0)$ and $\tau_1 = (\{\alpha_1 = \alpha_2 \to q, \alpha_2 = \alpha_1 \wedge \alpha_3 \to q, \alpha_3 = q\}, \alpha_1)$. $\tau_1 \leq \tau_0$ holds, with the relation $\{(\alpha_1, \alpha_0), (\alpha_0, \alpha_2)\}$ as a witness.

2.2 μHORS

We introduce below μHORS and its model checking problem, and reduce the latter to a type checking problem. To our knowledge, the notion of μHORS is new, but it is a subclass of the untyped HORS studied by Tsukada and Kobayashi [24], and the reduction from μHORS model checking to type checking (Theorem 1) is a corollary of the result of [24]. We shall therefore quickly go through the definitions and results; more formal definitions (apart from recursive types) and intuitions are found in [15, 9, 24].

μHORS and model checking problems The set of basic types (called *sorts*) is the subset of recursive intersection types, where Q is a singleton set $\{\mathtt{o}\}$ (where $\mathtt{o}$ is the type of trees) and there is no intersection: in $\sigma = \bigwedge\{\alpha_1, \dots, \alpha_k\}$, k is always 1. Below we often use the following term representation of sorts:

$$\kappa ::= \alpha \mid \kappa_1 \to \cdots \to \kappa_\ell \to \mathtt{o} \mid \mu\alpha.\kappa.$$

Let Σ be a ranked alphabet, i.e., a map from symbols to their arities. An element of $dom(\Sigma)$ is used as a tree constructor. A *sort environment* is a map from variables to sorts. The set of *applicative terms* of type κ under a sort environment $\mathcal{K}$ is inductively defined by the following rules:

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

$$\frac{\mathcal{K} \vdash t_1 : \kappa_1 \qquad \mathcal{K} \vdash t_2 : \kappa_2 \qquad \kappa_1 \cong (\kappa_2 \to \kappa)}{\mathcal{K} \vdash t_1\, t_2 : \kappa}$$

As usual, applications are left-associative, so that $t_1\, t_2\, t_3$ means $(t_1\, t_2)\, t_3$.

A μHORS $\mathcal{G}$ is a quadruple $(\mathcal{N}, \Sigma, \mathcal{R}, S)$ where: (i) $\mathcal{N}$ is a map from variables (called *non-terminals*) to sorts; (ii) Σ is a ranked alphabet, where $dom(\mathcal{N}) \cap dom(\Sigma) = \emptyset$; (iii) $\mathcal{R}$ is a map from non-terminals to a λ-term of the form $\lambda x_1. \cdots \lambda x_\ell.t$ where t is an applicative term; (iv) S, called the *start symbol*, is a non-terminal such that $\mathcal{N}(S) = \mathtt{o}$. If $\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_k \to \mathtt{o}$ and $\mathcal{R}(F) = \lambda x_1. \cdots \lambda x_\ell.t$, then it must be the case that $k = \ell$ and $\mathcal{N}, x_1{:}\kappa_1, \dots, x_\ell{:}\kappa_\ell \vdash t : \mathtt{o}$.

The (possibly infinite) tree generated by $\mathcal{G}$, written by $\mathit{Tree}(\mathcal{G})$, is defined as the limit of infinite fair reductions of S [15] where the reduction relation $\longrightarrow$ is defined by: (i) $F\, t_1 \cdots t_\ell \longrightarrow [t_1/x_1, \dots, t_\ell/x_\ell]t$ if $\mathcal{R}(F) = \lambda x_1. \cdots \lambda x_\ell.t$; and (ii) $a\, t_1 \cdots t_\ell \longrightarrow a\, t_1 \cdots t_{i-1}\, t_i'\, t_{i+1} \cdots t_\ell$ if $t_i \longrightarrow t_i'$ for some $i \in \{1, \dots, \ell\}$. See [15] for the formal definition of $\mathit{Tree}(\mathcal{G})$.

Notation 1 *We write $\widetilde{u}$ for a sequence $u_1 \cdots u_\ell$. $\lambda\widetilde{x}.t$ stands for $\lambda x_1. \cdots \lambda x_\ell.t$, and $[\widetilde{s}/\widetilde{x}]t$ for $[s_1/x_1, \dots, s_\ell/x_\ell]t$ (with the understanding that $\widetilde{s}$ and $\widetilde{x}$ have the same length ℓ). We often write the four components of $\mathcal{G}$ as $\mathcal{N}_\mathcal{G}, \Sigma_\mathcal{G}, \mathcal{R}_\mathcal{G}, S_\mathcal{G}$, and omit the subscript if it is clear from context. We often write $\mathcal{R}$ as a set of rewriting rules $\{F_1\, x_1 \cdots x_{\ell_1} \to t_1, \dots, F_m\, x_1 \cdots x_{\ell_m} \to t_m\}$ if $\mathcal{R}(F_i) = \lambda x_1. \cdots \lambda x_{\ell_i}.t_i$ for each $i \in \{1, \dots, m\}$.*

Example 2. Consider μHORS $\mathcal{G}_1 = (\mathcal{N}_1, \Sigma_1, \mathcal{R}_1, S)$ where $\mathcal{N}_1 = \{S \mapsto \mathtt{o}, F \mapsto (\mathtt{o} \to \mathtt{o})\}$, $\Sigma_1 = \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0\}$, and $\mathcal{R}_1 = \{S \to F\,\mathtt{c},\ \ F\,x \to \mathtt{a}\,x\,(F\,(\mathtt{b}\,x))\}$. S is rewritten as follows, and the tree in Figure 1 is generated: $S \longrightarrow F\,\mathtt{c} \longrightarrow \mathtt{a}\,\mathtt{c}\,(F\,(\mathtt{b}\,\mathtt{c})) \longrightarrow \mathtt{a}\,\mathtt{c}\,(\mathtt{a}\,(\mathtt{b}\,\mathtt{c})\,(F\,(\mathtt{b}\,(\mathtt{b}\,\mathtt{c})))) \longrightarrow \cdots$.

Example 3. Consider μHORS $\mathcal{G}_2 = (\mathcal{N}_2, \Sigma_1, \mathcal{R}_2, S)$ where Σ_1 is as given in Example 2, and: $\mathcal{N}_2 = \{S \mapsto \mathtt{o}, F \mapsto (\mathtt{o} \to \mathtt{o}), G \mapsto \mu\alpha.(\alpha \to \mathtt{o} \to \mathtt{o})\}$ and $\mathcal{R}_2 = \{S \to F\,\mathtt{c},\ F\,x \to G\,G\,x,\ G\,g\,x \to \mathtt{a}\,x\,(g\,g\,(\mathtt{b}\,x))\}$. This is the same as $\mathcal{G}_1$ except that recursive types are used instead of term-level recursion. S is reduced as below, and the same tree as $\mathit{Tree}(\mathcal{G}_1)$ is generated.

$$S \longrightarrow F\,\mathtt{c} \longrightarrow G\,G\,\mathtt{c} \longrightarrow \mathtt{a}\,\mathtt{c}\,(G\,G\,(\mathtt{b}\,\mathtt{c})) \longrightarrow \mathtt{a}\,\mathtt{c}\,(\mathtt{a}\,(\mathtt{b}\,\mathtt{c})\,(G\,G\,(\mathtt{b}\,(\mathtt{b}\,\mathtt{c})))) \longrightarrow \cdots$$

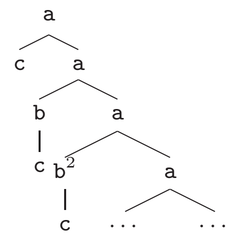

Fig. 1. The tree generated by $\mathcal{G}_1$ of Example 2.

Remark 1. A tree node that is never instantiated to a terminal symbol is expressed by the special terminal symbol $\perp$ (with arity 0). For example, for μHORS $\mathcal{G}_3 = (\mathcal{N}_3, \Sigma_1, \mathcal{R}_3, S)$ where $\mathcal{N}_3 = \{S \mapsto \mathtt{o}, F \mapsto \mu\alpha.(\alpha \to \mathtt{o})\}$ and $\mathcal{R}_3 = \{S \to F\,F,\ F\,x \to x\,x\}$, $\mathit{Tree}(\mathcal{G}_3)$ is a singleton tree $\perp$. □

As usual [15, 9], we use (top-down) tree automata to express properties of the tree generated by higher-order recursion schemes. For a ranked alphabet Σ, a *Σ-labeled tree* T is a map from sequences of natural numbers (which represent paths of the tree) to $\mathit{dom}(\Sigma)$, such that (i) its domain $\mathit{dom}(T)$ is non-empty and closed under the prefix operation, and (ii) if $\pi \in \mathit{dom}(T)$ then $\{j \mid \pi j \in \mathit{dom}(T)\} = \{1, \ldots, \Sigma(T(\pi))\}$. A (deterministic) *trivial automaton* $\mathcal{B}$ is a quadruple (Σ, Q, δ, q_0), where Σ is a ranked alphabet, Q is a finite set of states, δ, called a transition function, is a partial map from $Q \times \mathit{dom}(\Sigma)$ to Q^* such that $|\delta(q, a)| = \Sigma(a)$, and q_0 is the initial state. A Σ-labeled tree T is *accepted* by $\mathcal{B}$ if there is a Q-labeled tree R (called a *run tree*) such that: (i) $\mathit{dom}(T) = \mathit{dom}(R)$; (ii) $R(\epsilon) = q_0$; and (iii) for every $\pi \in \mathit{dom}(R)$, $\delta(R(\pi), T(\pi)) = R(\pi 1) \cdots R(\pi\Sigma(T(\pi)))$. For a trivial automaton $\mathcal{B} = (\Sigma, Q, \delta, q_0)$ (with $\perp \notin \mathit{dom}(\Sigma)$), we write $\mathcal{B}^\perp$ for the trivial automaton $(\Sigma \cup \{\perp \mapsto 0\}, Q, \delta \cup \{(q, \perp) \mapsto \epsilon) \mid q \in Q\}, q_0)$. We often write $\Sigma_\mathcal{B}, Q_\mathcal{B}, \delta_\mathcal{B}, q_{\mathcal{B},0}$ for the four components of $\mathcal{B}$, and omit the subscript if it is clear from context. Trivial automata are sufficient for describing safety properties: see [12] for the logical characterization.

Example 4. Let $\mathcal{B}_1 = (\Sigma_1, \{q_0, q_1\}, \delta, q_0)$ where Σ_1 is as given in Example 2 and δ is given by: $\delta(q_0, \mathtt{a}) = q_0 q_0$, $\delta(q_0, \mathtt{b}) = \delta(q_1, \mathtt{b}) = q_1$, and $\delta(q_0, \mathtt{c}) = \delta(q_1, \mathtt{c}) = \epsilon$. It accepts a Σ_1-labeled (ranked) tree T if and only if $\mathtt{a}$ does not occur below $\mathtt{b}$. In particular, $\mathcal{B}_1$ accepts the tree shown in Figure 1. □

The μHORS *model checking* is the problem of checking whether $\mathit{Tree}(\mathcal{G})$ is accepted by $\mathcal{B}^\perp$, given a μHORS $\mathcal{G}$ and a trivial automaton $\mathcal{B}$. The problem is in general undecidable [24]. We give a sound type system for checking that $\mathit{Tree}(\mathcal{G})$ is accepted by $\mathcal{B}^\perp$. The set of recursive intersection types is as given in Section 2.1, where the set Q of base types is the set of states of $\mathcal{B}$. Intuitively, a state q is regarded as the type of trees accepted by $\mathcal{B}^\perp$ from the state q [9].

The type judgment relations $\Gamma \vdash_\mathcal{B} t : \tau$ and $\Gamma \vdash_\mathcal{B} (\mathcal{G}, t) : \tau$ (where Γ, called a type environment, is a set of type bindings of the form $x : \tau$) are defined by:

$$\frac{\tau \leq \tau'}{\Gamma, x:\tau \vdash_{\mathcal{B}} x:\tau'} \qquad \frac{\delta_{\mathcal{B}}(q,a) = q_1 \cdots q_k \qquad q_1 \to \cdots \to q_k \to q \leq \tau}{\Gamma \vdash_{\mathcal{B}} a:\tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{B}} t_1 : \bigwedge_{i\in I} \tau_i \to \tau \\ \Gamma \vdash_{\mathcal{B}} t_2 : \tau_i' \text{ and } \tau_i' \leq \tau_i \text{ (for every } i \in I)\end{array}}{\Gamma \vdash_{\mathcal{B}} t_1 t_2 : \tau} \qquad \frac{\begin{array}{c}\Gamma, x:\tau_1, \ldots, x:\tau_\ell \vdash_{\mathcal{B}} t:\tau \\ x \text{ does not occur in } \Gamma\end{array}}{\Gamma \vdash_{\mathcal{B}} \lambda x.t : \bigwedge_{i\in\{1,\ldots,\ell\}} \tau_i \to \tau}$$

$$\frac{\forall (F:\tau) \in \Gamma.(\Gamma \vdash_{\mathcal{B}} \mathcal{R}(F):\tau)}{\vdash_{\mathcal{B}} \mathcal{R}:\Gamma} \qquad \frac{\vdash_{\mathcal{B}} \mathcal{R}_{\mathcal{G}}:\Gamma \qquad \Gamma \vdash_{\mathcal{B}} t:\tau}{\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, t):\tau}$$

The following theorem is a special case of the soundness of Tsukada and Kobayashi's infinite intersection type system for untyped HORS [24].

Theorem 1 (soundness). *Let $\mathcal{B}$ be a trivial automaton $(\Sigma, Q, \delta, q_{\mathcal{B},0})$ and $\mathcal{G}$ be a μHORS. If $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$, then Tree($\mathcal{G}$) is accepted by $\mathcal{B}^\perp$.*

Example 5. Recall $\mathcal{G}_1$ and $\mathcal{G}_2$ in Examples 2 and 3, and $\mathcal{B}_1$ in Example 4. $\Gamma_1 \vdash_{\mathcal{B}_1} (\mathcal{G}_1, S) : q_0$ and $\Gamma_2 \vdash_{\mathcal{B}_1} (\mathcal{G}_2, S) : q_0$ hold for $\Gamma_1 = \{S : q_0, F : (q_0 \wedge q_1) \to q_0\}$ and $\Gamma_2 = \Gamma_1 \cup \{G : \mu\alpha.(\alpha \to (q_0 \wedge q_1) \to q_0)\}$. □

Given a type environment Γ, a μHORS $\mathcal{G}$, and an automaton $\mathcal{B}$, it is decidable whether $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ holds. Thus, Γ can be used as a certificate for *Tree*($\mathcal{G}$) being accepted by $\mathcal{B}$. The converse of the theorem above does not hold, i.e., there is a μHORS $\mathcal{G}$ such that *Tree*($\mathcal{G}$) is accepted by $\mathcal{B}^\perp$ but *Tree*($\mathcal{G}$) is not well-typed. We have the following properties on the (un)decidability of type checking. See [10] for a proof.

Theorem 2. *1. Given a type environment Γ, a μHORS $\mathcal{G}$, and a trivial automaton $\mathcal{B}$, it is decidable whether $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ holds.*
2. Given a μHORS $\mathcal{G}$ and a trivial automaton $\mathcal{B}$, it is undecidable whether there exists Γ such that $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ holds.

3 Model Checking μHORS

We now describe the main result of this paper: a model checking procedure for μHORS. We shall develop a procedure Check that satisfies:

$$\textsc{Check}(\mathcal{G}, \mathcal{B}) = \begin{cases} \Gamma' \text{ such that } \Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0} \text{ if } \exists \Gamma.\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0} \\ \texttt{No} \text{ (with a counterexample) if } \textit{Tree}(\mathcal{G}) \text{ is not accepted by } \mathcal{B}^\perp \end{cases}$$

By Theorem 2, the procedure Check can only be a semi-algorithm: it may not terminate if *Tree*($\mathcal{G}$) is accepted by $\mathcal{B}^\perp$ but $\exists \Gamma.\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ does not hold.

An obvious approach would be to run (i) a sub-procedure FindCert($\mathcal{G}, \mathcal{B}$) to enumerate all the finite type environments Γ and output Γ if $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ holds, and in parallel, (ii) a sub-procedure FindCE($\mathcal{G}, \mathcal{B}$) to reduce $\mathcal{G}$ in a fair manner and output `No` if a partially generated tree is not accepted by $\mathcal{B}^\perp$. The first sub-procedure FindCert is, however, too non-deterministic to be used in practice.

We describe below a more realistic procedure for FINDCERT$(\mathcal{G}, \mathcal{B})$ that outputs Γ such that $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ if there is any, and may diverge otherwise. As FINDCERT can incrementally find the types of non-terminals, we can use them to improve FINDCE as well, by removing well-typed terms from the search space. As such interaction between FINDCERT and FINDCE is the same as the case without recursive types [8], we focus on the discussion of FINDCERT below.

3.1 Type Inference Procedure

We first give an informal overview of the idea of FINDCERT. Since it is easy to check whether a given Γ is a valid certificate (i.e. whether $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ holds), the main issue is how to find candidates for Γ. As in the algorithm for HORS without recursive types [8], the idea of finding Γ is to extract type information by partially reducing a given recursion scheme, and observing how each non-terminal symbol is used. For example, suppose that S is reduced as follows. $S : q_0 \longrightarrow^* C_1[F\,G : q_1] \longrightarrow^* C_2[G\,t : q_2] \longrightarrow^* C_3[t : q_1]$. Here, we have annotated each term with a state of the property automaton; $t : q$ means that the tree generated by t should be accepted from q. From the reduction sequence, we know t should have type q_1, from which we can guess that G should have type $q_1 \to q_2$, and we can further guess that F should have type $(q_1 \to q_2) \to q_1$. This way of guessing types is complete for HORS (without recursive types) [8]. In the presence of recursive types, however, we need a further twist, to obtain (relative) completeness. For example, suppose S is reduced as follows. $S : q_0 \longrightarrow^* C_1[F\,t_1 : q_1] \longrightarrow^* C_2[t_1\,t_2 : q_0] \longrightarrow^* C_3[t_2\,t_3 : q_1] \longrightarrow^* C_4[t_3\,t_4 : q_0] \longrightarrow^* \cdots$. This kind of calling chain terminates for ordinary HORS (since the terms are simply-typed), but may not terminate for μHORS because of recursive types. (For example, consider a variation of $\mathcal{G}_3$ in Remark 1, where the rule for F is replaced by $F\,x \to x\,(I\,x)$, with the new rule $I\,x \to x$. Then, we have an infinite calling chain: $S \longrightarrow^* F\,(I\,F) \longrightarrow^* (I\,F)\,(I\,(I\,F)) \longrightarrow^* (I\,(I\,F))\,(I\,(I\,(I\,F))) \longrightarrow^* \cdots$.) Thus, we would obtain an infinite set of type equations:
$\alpha_F = \alpha_{t_1} \to q_1 \quad \alpha_{t_1} = \alpha_{t_2} \to q_0 \quad \alpha_{t_2} = \alpha_{t_3} \to q_1 \quad \alpha_{t_3} = \alpha_{t_4} \to q_0 \quad \cdots$
(where α_t represents the type of term t). To address this problem, we introduce an equivalence relation $\sim$ on terms, and consider reductions modulo $\sim$. In the example above, if we choose $\sim$ so that $t_{2n-1} \sim t_{2n+1}$ and $F \sim t_{2n} \sim t_{2n+2}$, then we would have finite equations $\alpha_{[F]} = \alpha_{[t_1]} \to q_1$ and $\alpha_{[t_1]} = \alpha_{[F]} \to q_0$ (where $[t]$ is the equivalence class containing t), from which we can infer $\mu\alpha.(\alpha \to q_0) \to q_1$ as the type of F. As we show in Theorem 3 later, this way of type inference is complete *if a proper equivalence relation $\sim$ is given as an oracle*. It is not complete in general, but Theorem 4 ensures that no matter how $\sim$ is chosen, we can "amend" the inferred type environment to obtain a correct type environment. Based on the theorem, we can develop a complete procedure for FINDCERT.

We now turn to describe the idea more formally. Let $\mathbf{Tm}$ be the set of (well-sorted) closed terms constructed from non-terminals and terminals of $\mathcal{G}$, and $\sim$ be an equivalence relation on $\mathbf{Tm}$ that induces a *finite* set of equivalence classes. We write $[t]_\sim$ for the equivalence class containing t, i.e., $\{t' \mid t \sim t'\}$, and omit the subscript if clear from context. Intuitively, the equivalence relation $t_1 \sim t_2$

means that t_1 and t_2 behave similarly with respect to the given automaton $\mathcal{B}$. For the moment, we assume that $\sim$ is given as an oracle. Throughout the paper, we consider only equivalence relations that equate terms of the same sort, i.e., $t \sim t'$ implies $\mathcal{N} \vdash t : \kappa \Longleftrightarrow \mathcal{N} \vdash t' : \kappa$ for every κ.

We define the extended reduction relation $(\mathcal{X},\mathcal{U}) \longrightarrow_{\sim} (\mathcal{X}',\mathcal{U}')$ as the least relation closed under the rules below, where $\mathcal{X}$ is a set of terms and $\mathcal{U}$ is a set of pairs consisting of a term and an automaton state or a special element `fail`. In rule R-NT, $\mathbf{STm}(t)$ denotes the set of all subterms of t.

$$\frac{(a\,t_1\,\cdots\,t_\ell, q) \in \mathcal{U} \qquad \delta(q,a) = q_1 \cdots q_\ell}{(\mathcal{X},\mathcal{U}) \longrightarrow_{\sim} (\mathcal{X},\mathcal{U} \cup \{(t_1,q_1),\ldots,(t_\ell,q_\ell)\})} \qquad \text{(R-Const)}$$

$$\frac{(a\,t_1\,\cdots\,t_\ell, q) \in \mathcal{U} \qquad \delta(q,a) \text{ is undef. or } |\delta(q,a)| \neq \ell}{(\mathcal{X},\mathcal{U}) \longrightarrow_{\sim} (\mathcal{X},\mathcal{U} \cup \{\mathtt{fail}\})} \qquad \text{(R-F)}$$

$$\frac{(F\,\widetilde{t}, q) \in \mathcal{U} \qquad \mathcal{R}(F) = \lambda\widetilde{x}.u}{(\mathcal{X},\mathcal{U}) \longrightarrow_{\sim} (\mathcal{X} \cup \mathbf{STm}([\widetilde{t}/\widetilde{x}]u),\mathcal{U} \cup \{([\widetilde{t}/\widetilde{x}]u, q)\})} \qquad \text{(R-NT)}$$

$$\frac{(t\,t_1\,\cdots\,t_k, q) \in \mathcal{U} \qquad t \sim t' \qquad t' \in \mathcal{X}}{(\mathcal{X},\mathcal{U}) \longrightarrow_{\sim} (\mathcal{X},\mathcal{U} \cup \{(t'\,t_1\,\cdots\,t_k, q)\})} \qquad \text{(R-Eq)}$$

The main differences from the reduction relation $t \longrightarrow t'$ in Section 2.2 are: (i) each term t (of sort `o`) is coupled with its expected type, (ii) such pairs are kept in the $\mathcal{U}$ component after reductions (in other words, $(t,q) \in \mathcal{U}$ means that t should generate a tree accepted by $\mathcal{B}$ from state q), (iii) the $\mathcal{X}$ component keeps all the sub-terms that have occurred so far, and (iv) a subterm in a head position can be replaced by another term belonging to the same equivalence class (see rule R-Eq above). In rule R-Const, $(a\,t_1\,\cdots\,t_\ell, q)$ being an element of $\mathcal{U}$ means that $a\,t_1\,\cdots\,t_\ell$ should generate a tree of type q (i.e., should be accepted by $\mathcal{B}$ from the state q). The premise $\delta(q,a) = q_1 \cdots q_\ell$ means that the i-th subtree should have type q_i, so that we add (t_i, q_i) to the second component. Rule R-F is applied when $(a\,t_1\,\cdots\,t_\ell, q)$ is in the second set but no tree having a as its root can be accepted from the state q. The condition $|\delta(q,a)| \neq \ell$ actually never holds, by the assumption that $\sim$ equates only terms of the same sort. R-NT is the rule for reducing non-terminals. As mentioned above, rule R-Eq is used to replace a head of a term with an equivalent term with respect to $\sim$. Extended reduction sequences are in general infinite, and non-deterministic.

Example 6. Recall $\mathcal{G}_2$ in Example 3. Let $\sim^{(1)}$ be the least congruence relation that satisfies $\mathtt{b}(\mathtt{c}) \sim \mathtt{c}$. Then, by using $\sim^{(1)}$ as $\sim$, we can reduce $(\{S\}, \{(S, q_0)\})$ as follows:

$$\begin{tikzcd}
(S, q_0) \arrow[r] & (F\,c, q_0) \arrow[r] & (G\,G\,c, q_0) \arrow[r] & (a\,c\,(G\,G\,(b\,c)), q_0) \arrow[r] \arrow[d] & & (G\,G\,(b\,c), q_0) \arrow[dr] & & \\
 & & & (c, q_0) \arrow[rr, dashed, bend right] & (c, q_1) & (b\,c, q_0) \arrow[l] \arrow[ull, dashed, bend right] & (a\,(b\,c)\ \ (G\,G\,(b\,(b\,c))), q_0) \arrow[l] \arrow[r] & \cdots
\end{tikzcd}$$

Here, we have omitted the $\mathcal{X}$-component, and shown only elements relevant to reductions instead of the whole $\mathcal{U}$-component. In the figure, dashed arrows represent reductions by using rule R-Eq, and solid arrows represent reductions obtained by the other rules. From an infinite fair reduction sequence, we obtain the following set as $\mathcal{U}$:

$$\{(F\,(\mathtt{b}^k\,\mathtt{c}), q_0), (G\,G\,(\mathtt{b}^k\,\mathtt{c}), q_0), (\mathtt{b}^k\,\mathtt{c}, q_0), (\mathtt{b}^k\,\mathtt{c}, q_1) \mid k \geq 0\}$$
$$\cup\{(S, q_0)\} \cup \{(\mathtt{a}\,(\mathtt{b}^k\,\mathtt{c})\,(G\,G\,(\mathtt{b}^\ell\,\mathtt{c})), q_0) \mid k, \ell \geq 0\} \qquad \square$$

The goal below is to construct a candidate of type environment Γ that satisfies $\Gamma \vdash_{\mathcal{B}} \mathcal{G} : q_0$, from a fair reduction sequence (where a reduction sequence is fair if every enabled reduction is eventually reduced). The idea of the construction of Γ is similar to the case for ordinary HORS [8]. For example, in Example 6 above, from the pairs $(\mathtt{c}, q_0)$ and $(\mathtt{c}, q_1)$, we can guess that the type of $\mathtt{c}$ is $q_0 \wedge q_1$. From the pair $(F\,\mathtt{c}, q_0)$, we guess that the return type of F is q_0, so that the type of F is $q_0 \wedge q_1 \to q_0$. The actual construction is, however, more involved than [8] because of the presence of recursive types and the term equivalence relation $\sim$.

Let $(\mathcal{X}_0, \mathcal{U}_0) \longrightarrow_\sim (\mathcal{X}_1, \mathcal{U}_1) \longrightarrow_\sim \cdots$ be a fair reduction sequence where $\mathcal{X}_0 = \{S\}$ and $\mathcal{U}_0 = \{(S, q_0)\}$, and let $\mathcal{X}$ and $\mathcal{U}$ be $\bigcup_{i\in\omega} \mathcal{X}_i$ and $\bigcup_{i\in\omega} \mathcal{U}_i$ respectively. We prepare a type variable $\alpha_{[t_0],\ldots,[t_k],q}$ for each $(t_0 t_1 \cdots t_k, q) \in \mathcal{U}$. Intuitively, $\alpha_{[t_0],\ldots,[t_k],q}$ is the type of t_0 in $t_0\,t_1 \cdots t_k : q$. Let E be:

$$\{\alpha_{[t_0],[t_1],\ldots,[t_k],q} = \sigma_{[t_1]} \to \cdots \to \sigma_{[t_k]} \to q \mid (t_0\,t_1 \cdots t_k, q) \in \mathcal{U}\},$$

where $\sigma_{[t]} = \bigwedge\{\alpha_{[t],[t'_1],\ldots,[t'_\ell],q'} \mid (t\,t'_1 \cdots t'_\ell, q') \in \mathcal{U}\}$. We define the type environment $\Gamma_{\mathcal{X},\mathcal{U},\sim}$ as $\{F : (E, \alpha_{[F],[t_1],\ldots,[t_k],q}) \mid (F\,t_1 \cdots t_k, q) \in \mathcal{U}\}$. By the condition that $\sim$ induces a *finite* number of equivalence classes, $\Gamma_{\mathcal{X},\mathcal{U},\sim}$ is finite.

Example 7. From the reductions in Example 6, we get the following type equations:

$$\alpha_{S,q_0} = q_0 \qquad \alpha_{F,\mathtt{c},q_0} = \alpha_{\mathtt{c},q_0} \wedge \alpha_{\mathtt{c},q_1} \to q_0 \qquad \alpha_{\mathtt{c},q_0} = q_0 \qquad \alpha_{\mathtt{c},q_1} = q_1$$
$$\alpha_{G,G,\mathtt{c},q_0} = \alpha_{G,G,\mathtt{c},q_0} \to \alpha_{\mathtt{c},q_0} \wedge \alpha_{\mathtt{c},q_1} \to q_0$$

Thus, the extracted type environment (in the usual term representation) is:

$$\{S : q_0, F : (q_0 \wedge q_1) \to q_0, G : \mu\alpha.(\alpha \to (q_0 \wedge q_1) \to q_0)\}. \qquad \square$$

The theorem below (see [10] for a proof) ensures that if $\mathcal{G}$ is typable and if $\sim$ is properly chosen, $\Gamma_{\mathcal{X},\mathcal{U},\sim}$ is a proper witness. For a type environment Γ, we define the equivalence relation $\sim_\Gamma$ by: $\sim_\Gamma = \{(t_1, t_2) \mid \forall\tau.(\Gamma \vdash t_1 : \tau \iff \Gamma \vdash t_2 : \tau)\}$.

Theorem 3. *If $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ and $\sim \subseteq \sim_\Gamma$, then $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$.*

Example 8. Recall $\mathcal{G}_2$ in Example 3, and $\Gamma_2 = \{S : q_0, F : (q_0 \wedge q_1) \to q_0, G : \mu\alpha.(\alpha \to (q_0 \wedge q_1) \to q_0)\}$ in Example 5. The relation $\sim$ in Example 6 satisfies the assumption $\sim \subseteq \sim_{\Gamma_2}$ of Theorem 3, and $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}_1} (\mathcal{G}_2, S) : q_0$ holds indeed.

Theorem 3 cannot be directly used for type inference, since we do not know $\sim_\Gamma$ in advance. We shall prove below (in Theorem 4) that even if $\sim$ is not a subset of $\sim_\Gamma$, we can "amend" the type environment to get a valid one, by using the refinement relation $\sqsubseteq$ below. Intuitively, $\tau_1 \sqsubseteq \tau_2$ means that τ_1 is obtained from τ_2 by removing some intersection types. Note that unlike subtyping, the refinement relation is co-variant in the function type constructor ($\to$).

Definition 3 (refinement). *Let $\tau_0 = (E_0, \alpha_0)$ and $\tau_1 = (E_1, \alpha_1)$ be closed types, and let $E = E_0 \cup E_1$. The type τ_0 is a* refinement *of τ_1, written $\tau_0 \sqsubseteq \tau_1$, if there exists a binary relation $\mathcal{R}$ on $\mathbf{Tv}(\tau_1) \cup \mathbf{Tv}(\tau_2)$ such that (i) $(\tau_0, \tau_1) \in \mathcal{R}$ and (ii) for every $(\tau_0', \tau_1') \in \mathcal{R}$, there exist $\sigma_1, \dots, \sigma_m, \sigma_1', \dots, \sigma_m', q$ such that $E(\alpha_0') = \sigma_1 \to \cdots \to \sigma_m \to q$ and $E(\alpha_1') = \sigma_1' \to \cdots \to \sigma_m' \to q$, with $(\sigma_1, \sigma_1'), \dots, (\sigma_m, \sigma_m') \in \mathcal{R}^{\sqsubseteq}$. Here, $\mathcal{R}^{\sqsubseteq}$ is defined as:*
$\{(\bigwedge\{\alpha_1, \dots, \alpha_k\}, \bigwedge\{\alpha_1', \dots, \alpha_{k'}'\}) \mid \forall i \in \{1, \dots, k\}. \exists j \in \{1, \dots, k'\}. \alpha_i \mathcal{R} \alpha_j'\}$.

We write $\mathit{\Gamma}_1 \sqsubseteq \mathit{\Gamma}_2$ if $dom(\mathit{\Gamma}_1) \subseteq dom(\mathit{\Gamma}_2)$ and for every $x : \tau_1 \in \mathit{\Gamma}_1$, there exists τ_2 such that $x : \tau_2 \in \mathit{\Gamma}_2$ and $\tau_1 \sqsubseteq \tau_2$.

Example 9. Let τ_1 be $q_1 \to q_2$ and τ_2 be $(q_1 \wedge q_0) \to q_2$. Then $\tau_1 \sqsubseteq \tau_2$ and $\tau_1 \to q_0 \sqsubseteq \tau_2 \to q_0$ hold. Note that $\tau_1 \leq \tau_2$ but $\tau_1 \to q_0 \not\leq \tau_2 \to q_0$. □

Theorem 4. *Suppose $\mathit{\Gamma} \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$. Let $\sim$ be an equivalence relation on* $\mathbf{Tm}$ *and $(\mathcal{X}_0, \mathcal{U}_0) \longrightarrow_\sim (\mathcal{X}_1, \mathcal{U}_1) \longrightarrow_\sim (\mathcal{X}_2, \mathcal{U}_2) \longrightarrow_\sim \cdots$ be a fair reduction sequence, with $(\mathcal{X}_0, \mathcal{U}_0) = (\{S\}, \{(S, q_{\mathcal{B},0})\})$. Let $\mathcal{U} = \bigcup_i \mathcal{U}_i$ and $\mathcal{X} = \bigcup_i \mathcal{X}_i$. Then, there exists $\mathit{\Gamma}'$ such that $\mathit{\Gamma}' \sqsubseteq \mathit{\Gamma}_{\mathcal{X},\mathcal{U},\sim}$ and $\mathit{\Gamma}' \vdash (\mathcal{G}, S) : q_{\mathcal{B},0}$,*

The proof is given in the extended version [10]. Intuitively, Theorem 4 holds because, if $\sim$ is not a subset of $\sim_{\mathit{\Gamma}}$, we only get extra reduction sequences, whose effect is only to add extra type bindings and elements in intersections. Thus, by removing the extra nodes and edges (using the refinement relation from right to left), we can obtain a proper type environment.

Example 10. Recall Example 6. Let $\sim^{(2)}$ be $\sim^{(1)} \cup \{(G\,G, \mathtt{b}), (\mathtt{b}, G\,G)\}$. In addition to the reductions in Example 6, we obtain the extra reduction sequence: $(\mathtt{b}\,\mathtt{c}, q_1) \to (G\,G\,\mathtt{c}, q_1) \to (\mathtt{a}\,\mathtt{c}\,(G\,G\,(\mathtt{b}\,\mathtt{c})), q_1) \to \mathtt{fail}$. From the reductions, we obtain the following type equations:

$$\begin{array}{llll}
\alpha_{S,q_0} = q_0 & \alpha_{F,\mathtt{c},q_0} = \alpha_{\mathtt{c},q_0} \wedge \alpha_{\mathtt{c},q_1} \to q_0 & \alpha_{\mathtt{c},q_0} = q_0 & \alpha_{\mathtt{c},q_1} = q_1 \\
\multicolumn{4}{l}{\alpha_{G,G,\mathtt{c},q_0} = \alpha_{G,G,\mathtt{c},q_0} \underline{\wedge \alpha_{G,G,\mathtt{c},q_1}} \to \alpha_{\mathtt{c},q_0} \wedge \alpha_{\mathtt{c},q_1} \to q_0} \\
\multicolumn{4}{l}{\underline{\alpha_{G,G,\mathtt{c},q_1} = \alpha_{G,G,\mathtt{c},q_1} \wedge \alpha_{G,G,\mathtt{c},q_1} \to \alpha_{\mathtt{c},q_0} \wedge \alpha_{\mathtt{c},q_1} \to q_1}}
\end{array}$$

The part obtained from the extra reduction sequence is underlined. By ignoring that part, we get the same equations as Example 7, hence obtaining the correct type environment: $\{S : q_0, F : (q_0 \wedge q_1) \to q_0, G : \mu\alpha.(\alpha \to (q_0 \wedge q_1) \to q_0)\}$. □

Theorem 4 yields the procedure FINDCERT in Figure 2. The condition $\exists \mathit{\Gamma}'. \mathit{\Gamma}' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0} \wedge \mathit{\Gamma}' \sqsubseteq \mathit{\Gamma}$ is in general undecidable because of the presence of recursive types. Thus, we bound the size (i.e., the number of type constructors) of $\mathit{\Gamma}'$ by v, and gradually increase the bound. An algorithm to check whether there exists $\mathit{\Gamma}'$ such that $|\mathit{\Gamma}'| < v$ and $\mathit{\Gamma}' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0} \wedge \mathit{\Gamma}' \sqsubseteq \mathit{\Gamma}$ is discussed in Section 3.2. By Theorem 4, we have:

Theorem 5 (relative completeness). *If $\mathit{\Gamma} \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ for some finite recursive type environment $\mathit{\Gamma}$, then* FINDCERT$(\mathcal{G}, S, q_{\mathcal{B},0})$ *eventually terminates and outputs $\mathit{\Gamma}'$ such that $\mathit{\Gamma}' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$.*

To see the termination, notice that by the condition that $\sim$ induces a finite number of equivalence classes, there exists m such that $\mathit{\Gamma}_{\mathcal{X},\mathcal{U},\sim} = \mathit{\Gamma}_{\mathcal{X}_m,\mathcal{U}_m,\sim}$ in Theorem 4.

```
FINDCERT($\mathcal{G},\mathcal{B}$) = Rep($\mathcal{G},\mathcal{B},\{S\},\{(S,q_{\mathcal{B},0})\},\{(S,S)\},1$)
Rep($\mathcal{G},\mathcal{B},\mathcal{X},\mathcal{U},\sim,v$) =
  let ($\mathcal{X}$,$\mathcal{U}$) $\longrightarrow^{\ell}_{\sim}$ ($\mathcal{X}'$,$\mathcal{U}'$) in let $\sim' =$ expandEq($\sim,\mathcal{X}'$) in
   if $\Gamma'\vdash_{\mathcal{B}}(\mathcal{G},S){:}q_{\mathcal{B},0}$ for some $\Gamma'\sqsubseteq\Gamma_{\mathcal{X}',\mathcal{U}',\sim'}$ and $|\Gamma'|\leq v$ then return $\Gamma'$
   else Rep($\mathcal{G},\mathcal{B},\mathcal{X}',\mathcal{U}',\sim',v+1$)
```

Fig. 2. A type inference procedure. ($|\Gamma|$ denotes the largest type size in Γ.)

3.2 Type Checking by SAT Solving

We now discuss the sub-algorithm for FINDCERT, to check whether there exists Γ' such that $|\Gamma'| \leq v$ and $\Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0} \wedge \Gamma' \sqsubseteq \Gamma$.

We first rephrase the condition $|\Gamma'| \leq v \wedge \Gamma' \sqsubseteq \Gamma$. For a set $E = \{\alpha_1 = \sigma_{1,1} \to \cdots \sigma_{1,m_1} \to q_1, \ldots, \alpha_n = \sigma_{n,1} \to \cdots \sigma_{n,m_n} \to q_n\}$, we write $E^{(k)}$ for:

$$\{\alpha_1^{(1)} = \sigma_{1,1}^{(k)} \to \cdots \sigma_{1,m_1}^{(k)} \to q_1, \ldots, \alpha_n^{(1)} = \sigma_{n,1}^{(k)} \to \cdots \sigma_{n,m_n}^{(k)} \to q_n, \ldots,$$
$$\alpha_1^{(k)} = \sigma_{1,1}^{(k)} \to \cdots \sigma_{1,m_1}^{(k)} \to q_1, \ldots, \alpha_n^{(k)} = \sigma_{n,1}^{(k)} \to \cdots \sigma_{n,m_n}^{(k)} \to q_n, \},$$

obtained by preparing k copies for each type variable. Here, for $\sigma = \bigwedge\{\alpha_1, \ldots, \alpha_\ell\}$, $\sigma^{(k)}$ represents $\bigwedge\{\alpha_1^{(1)}, \ldots, \alpha_\ell^{(1)}, \ldots, \alpha_1^{(k)}, \ldots, \alpha_\ell^{(k)}\}$. Clearly, $(E, \alpha_i) \cong (E^{(k)}, \alpha_i^{(1)})$. We write $\Gamma^{(k)}$ for $\{x : (E^{(k)}, \alpha^{(i)}) \mid x : (E, \alpha) \in \Gamma, 1 \leq i \leq k\}$.

We write $E \sqsubseteq_{\mathtt{s}} E'$ if E is obtained from E' by removing some elements from intersections, i.e., if $E = \{\alpha_1 = \bigwedge S_{1,1} \to \cdots \bigwedge S_{1,m_1} \to q_1, \ldots, \alpha_n = \bigwedge S_{n,1} \to \cdots \bigwedge S_{n,m_n} \to q_n\}$ and $E' = \{\alpha_1 = \bigwedge S'_{1,1} \to \cdots \bigwedge S'_{1,m_1} \to q_1, \ldots \alpha_n = \bigwedge S'_{n,1} \to \cdots \bigwedge S'_{n,m_n} \to q_n\}$ with $S_{i,j} \subseteq S'_{i,j}$ for every i, j. It is pointwise extended to $\Gamma \sqsubseteq_{\mathtt{s}} \Gamma'$ by: $\Gamma \sqsubseteq_{\mathtt{s}} \Gamma' \iff \forall x : (E, \alpha) \in \Gamma, \exists x : (E', \alpha) \in \Gamma'.E \sqsubseteq_{\mathtt{s}} E'$. Then, $\Gamma' \sqsubseteq \Gamma$ is equivalent to $\exists k.\Gamma' \sqsubseteq_{\mathtt{s}} \Gamma^{(k)}$ (up to renaming of type variables). Thus, the condition $|\Gamma'| \leq v \wedge \Gamma' \sqsubseteq \Gamma$ in the algorithm can be replaced by $\Gamma' \sqsubseteq_{\mathtt{s}} \Gamma^{(v)}$ without losing completeness.

To check whether there exists Γ' such that $\Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ and $\Gamma' \sqsubseteq_{\mathtt{s}} \Gamma^{(k)}$, we attach a boolean variable to each type binding and each element of an intersection in $\Gamma^{(k)}$, to express whether Γ' has the corresponding binding or element. Thus, an annotated type environment is of the form $\{x_1 {:}^{b_1} \tau_1, \ldots, x_m {:}^{b_m} \tau_m\}$, where each type equation in $\tau_1, \ldots, \tau_m$ is now of the form:
$\alpha = \bigwedge_{i \in I_1} b_{1,i}\alpha_{1,i} \to \cdots \to \bigwedge_{k \in I_k} b_{k,i}\alpha_{k,i} \to q$.
Given an assignment function f for boolean variables, the type environment $f(\Delta)$ is given by:

$$\begin{array}{l}
f(\Delta) = \{x_i : f(\rho_i) \mid x_i :^{b_i} \rho_i \in \Delta \wedge f(b_i) = \mathtt{true}\} \\
f(E, \alpha) = (\{\alpha = f(\xi_1) \to \cdots \to f(\xi_k) \to q \mid (\alpha = \xi_1 \to \cdots \to \xi_k \to q) \in E\}, \alpha) \\
f(\bigwedge_{i \in I} b_i\alpha_i) = \bigwedge\{\alpha_i \mid i \in I, f(b_i) = \mathtt{true}\}
\end{array}$$

Let Δ be the type environment obtained by attaching boolean variables to $\Gamma^{(k)}$. Then, the condition $\Gamma' \sqsubseteq_{\mathtt{s}} \Gamma^{(k)} \wedge \Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ is reduced to: "Is there a boolean assignment f such that $f(\Delta) \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$?" It can be expressed as a SAT problem as follows. We first introduce additional boolean variables: (i) For each rule $F \mapsto \lambda x_1.\cdots\lambda x_k.t \in \mathcal{R}$, a subterm s of t, a type binding $F :^b \xi_1 \to \cdots \to \xi_k \to q \in \Delta$, and a type ρ in Δ, we prepare a variable $b_{\Delta, x_1:\xi_1,\dots,x_k:\xi_k \vdash s:\rho}$, which expresses whether $f(\Delta, x_1\!:\!\xi_1, \dots, x_k\!:\!\xi_k) \vdash_{\mathcal{B}} s : f(\rho)$ should hold. (ii) For each pair (ρ_1, ρ_2) of types occurring in Δ, we introduce $b_{\rho_1 \le \rho_2}$, which expresses whether $f(\rho_1) \le f(\rho_2)$ should hold. Now, the existence of a boolean assignment function f such that $f(\Delta) \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ is reduced to the satisfiability of the conjunction of all the following boolean formulas. We write $F\!:\!\bigwedge_{j\in\{1..n\}} b_j\rho_j \in \Delta$ for $F :^{b_1} \rho_1, \dots, F :^{b_n} \rho_n \in \Delta$ below. For simplicity, we omit type equations E and identify α and $E(\alpha)$ below.

(i) $\bigvee\{b_i \mid S :^{b_i} q_{\mathcal{B},0} \in \Delta\}$.
(ii) $b \Rightarrow b_{\Delta, x_1:\xi_1,\dots,x_k:\xi_k \vdash t:q}$
for each $F :^b \xi_1 \to \cdots \to \xi_k \to q \in \Delta$ such that $\mathcal{R}(F) = \lambda x_1, \dots, x_k.t$.
(iii) $b_{\Delta', x:\bigwedge_{j\in J} b_j\rho_j \vdash x:\rho} \Rightarrow \bigvee_{j\in J}(b_j \wedge b_{\rho_j \le \rho})$, for each $b_{\Delta', x:\bigwedge_{j\in J} b_j\rho_j \vdash x:\rho}$.
(iv) $b_{\Delta'\vdash a:\rho} \Rightarrow \bigvee\{b_{q_1\to\cdots\to q_k \to q \le \rho} \mid \delta(a,q) = q_1\cdots q_k\}$, for each $b_{\Delta'\vdash a:\rho}$.
(v) $b_{\Delta'\vdash t_1 t_2:\rho} \Rightarrow \bigvee(b_{\Delta'\vdash t_1:(\bigwedge_{j\in J} b_j\rho_j)\to\rho} \wedge (\bigwedge_{j\in J}(b_j \Rightarrow \bigvee(b_{\Delta'\vdash t_2:\rho'} \wedge b_{\rho'\le\rho_j}))))$,
for each $b_{\Delta'\vdash t_1 t_2:\rho}$.
(vi) $b_{(\bigwedge_{i\in I} b_i\rho_i)\le(\bigwedge_{j\in J} b_j\rho'_j)} \Rightarrow \bigwedge_{j\in J}(b_j \Rightarrow \bigvee_{i\in I}(b_i \wedge b_{\rho_i \le \rho'_j}))$,
for each $b_{(\bigwedge_{i\in I} b_i\rho_i)\le(\bigwedge_{j\in J} b_j\rho'_j)}$.
(vii) $b_{\xi_1\to\cdots\to\xi_k\to q \le \xi'_1\to\cdots\to\xi'_m\to q'} \Rightarrow k = m \wedge q = q' \wedge \bigwedge_{i\in\{1,\dots,k\}} b_{\xi'_i \le \xi_i}$,
for each $b_{\xi_1\to\cdots\to\xi_k\to q \le \xi'_1\to\cdots\to\xi'_m\to q'}$.

The first condition (i) ensures that $S\!:\!q_{\mathcal{B},0} \in f(\Delta)$. The condition (ii) ensures that each type binding in $f(\Delta)$ is valid (i.e., $\vdash_{\mathcal{B}} \mathcal{R} : f(\Delta)$). The next three conditions (iii)-(v) express the validity of a type judgment $f(\Delta, x_1\!:\!\xi_1, \dots, x_k\!:\!\xi_k) \vdash_{\mathcal{B}} t : f(\rho)$, corresponding to the typing rules for variables, constants, and applications. The last two conditions express the validity of a subtype relation.

By the above construction, there exists a boolean assignment function f such that $f(\Delta) \vdash (\mathcal{G}, S) : q_{\mathcal{B},0}$ if and only if the conjunction of the above boolean formulas is satisfiable. The latter can be solved by using a SAT solver.

Example 11. Recall $\Gamma_{\mathcal{X}^{(2)},\mathcal{U}^{(2)},\sim^{(2)}}$ in Example 10. By adding it with boolean variables, we obtain $\Delta = \{S : q_0, F : (q_0 \wedge q_1) \to q_0, G :^{b_0} \tau_0, G :^{b_1} \tau_1)\}$, where $\tau_i = (b_2\tau_0 \wedge b_3\tau_1) \to (q_0 \wedge q_1) \to q_i$ for $i \in \{0,1\}$. (Here, for the sake of simplicity, we have added boolean variables only to critical parts.) From the typing of G, we get the following boolean constraints:

$$b_i \Rightarrow b_{\Delta'\vdash \mathtt{a}\,x\,(g\,g\,(\mathtt{b}\,x)):q_i} \text{ (for } i \in \{0,1\}) \qquad b_{\Delta'\vdash \mathtt{a}\,x\,(g\,g\,(\mathtt{b}\,x)):q_0} \Rightarrow b_{\Delta'\vdash g\,g:q_0\wedge q_1\to q_0}$$
$$b_{\Delta'\vdash \mathtt{a}\,x\,(g\,g\,(\mathtt{b}\,x)):q_1} \Rightarrow \mathtt{false} \qquad b_{\Delta'\vdash g:\tau_0} \Rightarrow b_2 \qquad b_{\Delta'\vdash g:\tau_1} \Rightarrow b_3$$
$$b_{\Delta'\vdash g\,g:q_0\wedge q_1\to q_0} \Rightarrow (b_{\Delta'\vdash g:\tau_0} \wedge (b_2 \Rightarrow b_{\Delta'\vdash g:\tau_0}) \wedge (b_3 \Rightarrow b_{\Delta'\vdash g:\tau_1}))$$

Here, $\Delta' = \Delta \cup \{g :^{b_2} \tau_0, g :^{b_3} \tau_1, x : q_0, x : q_1\}$. The above conditions are satisfied by f such that $f(b_0) = f(b_2) = \mathtt{true}$ and $f(b_1) = f(b_3) = \mathtt{false}$. Thus, we get $\Gamma' = f(\Delta) = \{S : q_0, F : (q_0 \wedge q_1) \to q_0, G : \tau_0\}$ where $\tau_0 = \tau_0 \to (q_0 \wedge q_1) \to q_0$. We have $\Gamma' \vdash_{\mathcal{B}_1} (\mathcal{G}_2, S) : q_0$ as required. $\square$

4 Applications

This section discusses two applications of μHORS model checking: verification of (functional) object-oriented programs and that of higher-order multi-threaded programs. Those programs can be verified via reduction to μHORS model checking. In both applications, the translation from a source program to μHORS is just like giving the semantics of the source program (in terms of the λ-calculus). This comes from the expressive power of the model of μHORS model checking (i.e., μHORS), which is the main advantage of our approach.

4.1 Model-Checking Functional Objects

In this section, we discuss how to reduce verification problems for (functional) object-oriented programs. The idea is to transform a program into μHORS that generates a tree representing all the possible action sequences[3] of the source program. The translation is sound *and complete* in the sense that all *and only* action sequences that occur are represented in the tree. Properties that can be checked include: reachability (i.e., whether program execution reaches certain program points), order of method invocations, and whether downcasts may fail. In a full version [10], we give a formal translation from (a call-by-value variant of) Featherweight Java (FJ) [5] to μHORS.

We use the following classes that represent natural numbers with methods for addition (`add`) and predecessors (`pred`) as a running example. The statement `fail;` signals a global action that denotes a failure. Method `rand` non-deterministically returns a natural number that is equal to or greater than the argument; $\Box$ denotes a non-deterministic choice operator. The main expression to be executed takes a predecessor of a (non-deterministically chosen) non-zero natural number.

```
class Nat extends Object {
  Nat add(Nat n) { fail; }
  Nat pred() { fail; }
  Nat rand(Nat n) { return n□new S(this.rand(n)); } }
class Z extends Nat { Nat add(Nat n) { return n; } }
class S extends Nat {
  Nat p;
  Nat add(Nat n) { Nat p' = this.p.add(n);  return new S(p'); }
  Nat pred() { return this.p; } }
// main expression
new Z().rand(new Z()).add(new S(new Z())).pred();
```

To verify program execution does not `fail`, we translate the program to a μHORS that generates the tree representing all the possible global events, like: `br e (br e` $\cdots$). Here, `br` and `e` represent non-deterministic branch (caused by

[3] The source language has a construct to signal a global action, as well as a non-deterministic choice operator.

$\Box$) and program termination, respectively. Then, it suffices to check that the tree does not contain `fail` by using μHORS model checking.

Translation to μHORS. The main ideas of translation are: (i) to express an object as a record (or tuple) of functions that represent methods [2], and (ii) to represent each method in the continuation passing style (CPS) in order to correctly reflect the evaluation order and action sequences to μHORS. For example, an object of class `S` is expressed by a tuple $\langle S\_add, S\_pred, S\_rand\rangle$ of functions S_add, S_pred and S_rand that represent methods `add`, `pred`, and `rand` defined or inherited in class `S`, respectively.[4] A function that represents a method takes an argument that represents "self" and a continuation argument, as well as ordinary arguments of the method. In general, a method of the form

```
C0 m(C1 x1, ..., Cn xn) { return e; }
```

is represented by the λ-term $\lambda x_1.\cdots\lambda x_n.\lambda this.\lambda k.\,[\![\, e]\!]_k$ where k is the continuation parameter and $[\![e]\!]_k$ denotes the translation of e, which passes the result of method execution to k. For example, method `add` in class `S` is represented by non-terminal S_add, whose body is

$$\lambda n.\lambda this.\lambda k.\,[\![\,\texttt{Nat p' = ...; return new S(p');}]\!]_k$$

Then, method invocation is expressed as self-application [7]. For example, invocation of `add` on an `S` object with a `Z` object as an argument is expressed by

$$S\_add\ \langle Z\_add, Z\_pred, Z\_rand\rangle\ \langle S\_add, S\_pred, S\_rand\rangle\ k$$

where k is the current continuation. Note that S_add is applied to a tuple that contains itself.

To deal with fields, each method is further abstracted by values of fields of `this`. So, the body of S_add is in fact

$$\lambda p.\lambda n.\lambda this.\lambda k.\,[\![\,\texttt{Nat p' = ...; return new S(p');}]\!]_k,$$

where p stands for `this.p` inside the method body. Although this scheme only supports field access of the form `this.f`, field access to any expressions other than `this` can be expressed by using "getter" methods. A non-terminal representing a method will be applied to initial field values when an object is instantiated. For example, object instantiation `new S(p')` is represented by $\langle S\_add\ p', S\_pred\ p', S\_rand\ p'\rangle$. By using pattern-matching for λ, method `add` in class `S` is expressed by the following two rules:

$$\begin{aligned} S\_add &\mapsto \lambda\langle p_a, p_p, p_r\rangle.\lambda\langle n_a, n_p, n_r\rangle.\lambda\langle this_a, this_p, this_r\rangle.\lambda k. \\ &\qquad p_a\ \langle n_a, n_p, n_r\rangle\ \langle p_a, p_p, p_r\rangle\ (F\ k), \\ F &\mapsto \lambda k'.\lambda\langle p'_a, p'_p, p'_r\rangle. \\ &\qquad k'\ \langle S\_add\ \langle p'_a, p'_p, p'_r\rangle, S\_pred\ \langle p'_a, p'_p, p'_r\rangle, S\_rand\ \langle p'_a, p'_p, p'_r\rangle\rangle \end{aligned}$$

[4] μHORS does not have tuples as primitives but all the tuples can be eliminated. Tuples as function arguments can be eliminated by currying and there is no function that returns tuples, thanks to the CPS representation. See the full version for details.

where F stands for the continuation of the variable definition `Nat p' = ...;`.

A global action a such as `fail` is represented by a tree node a; non-deterministic choice is by the node `br` of arity 2. The (translation of the) main expression is given as the initial continuation a constant function that returns the tree node `e` of arity 0. So, in order to verify that the program does not `fail`, it suffices to verify that the generated tree consists only of nodes `br` and `e`.

We address the problem of the lack of subtyping in μHORS as follows. We represent every object as a tuple of the *same length* ℓ, where ℓ is the number of the methods defined in the whole program. If a certain method is undefined, we just insert a dummy function $\lambda\widetilde{x}.\lambda k.\mathtt{fail}$ in the corresponding position of the tuple. The dummy function just outputs `fail` to signal `NoSuchMethodError` whenever it is called.

The resulting encoding of an object is well-typed. Let $\{m_1, \dots, m_\ell\}$ be all the method names in the program, and $\{n_1, \dots, n_\ell\}$ be their arities. Then, the encoding of every object would have the same recursive sort κ_o, given by:

$$\kappa_o = \kappa_{m_1} \times \cdots \times \kappa_{m_\ell} \qquad \kappa_{m_i} = \underbrace{\kappa_o \to \cdots \kappa_o}_{n_i} \to \kappa_o \to \underbrace{(\kappa_o \to \mathtt{o})}_{\text{type of continuation}} \to \mathtt{o}$$

The source program execution yields a sequence of global actions $a_1 a_2 \cdots a_n$ if and only if the tree generated by the translation has a path labeled with $a_1 a_2 \cdots a_n$ (ignoring `br`). Thus safety property verification of FJ programs is reduced to μHORS model checking.

4.2 Model-Checking Higher-Order Multi-Threaded Programs

This section discusses how to apply the extended HO model checking to verification of multi-threaded programs, where each thread may use higher-order functions and recursion. For the sake of simplicity, we discuss only programs consisting of two threads, whose syntax is given by:

$$\begin{aligned} P \text{ (programs) } &::= M_1 \,||\, M_2 \\ M \text{ (threads) } &::= (\,) \mid a \mid x \mid \mathtt{fun}(f, x, M) \mid M_1 M_2 \mid M_1 \Box M_2 \end{aligned}$$

A program $P = M_1 \,||\, M_2$ executes two threads M_1 and M_2 concurrently, where M_1 and M_2 are (call-by-value) higher-order functional programs with side effects. The expression a performs a global action a, and evaluates to the unit value $(\,)$. We keep global actions abstract, so that various synchronization primitives and shared memory can be modeled. The expression $\mathtt{fun}(f, x, M)$ describes a recursive function f such that $f(x) = M$. When f does not occur in M, we write $\lambda x.M$ for $\mathtt{fun}(f, x, M)$. We also write **let** $x = M_1$ **in** M_2 for $(\lambda x.M_2)M_1$, and further abbreviate it to $M_1; M_2$ when x does not occur in M_2. $M_1 \Box M_2$ evaluates M_1 or M_2 non-deterministically. The formal semantics is given in [10]. The goal of verification is, given a program M and a property ψ on global action sequences, to check whether all the possible action sequences of M satisfy ψ.

Example 12. Let M be the following thread:

$$\mathbf{let}\ \mathit{sync}\ f = \texttt{lock}; f(); \texttt{unlock}; \mathit{sync}\ f\ \mathbf{in}\ \mathbf{let}\ \mathit{cr}\ x = \texttt{enter}; \texttt{exit}\ \mathbf{in}\ \mathit{sync}\ \mathit{cr},$$

which models a thread acquiring a global lock before entering a critical section. We may then wish to verify that the global actions `enter` and `exit` can occur only alternately, as long as `lock` and `unlock` occur alternately. □

We can reduce verification problems for multi-threads to extended HO model checking problems by transforming a given program to a μHORS that generates a tree describing all the possible global action sequences. The ideas of the transformation are: (i) transform each thread to CPS (continuation-passing style) to correctly model the order of actions, as in [9], and (ii) apply each thread to a scheduler, and let a thread pass the control to the scheduler non-deterministically after each global action. The translation from programs to μHORS is:

$$(M_1 \,||\, M_2)^\dagger = \texttt{br}\ (\mathit{Sched}\ ({M_1}^\dagger\ \lambda x.\texttt{e})\ ({M_2}^\dagger\ \lambda x.\texttt{e}))\ (\mathit{Sched}\ ({M_2}^\dagger\ \lambda x.\texttt{e})\ ({M_1}^\dagger\ \lambda x.\texttt{e}))$$
$$(\,)^\dagger = \lambda k.\lambda g.k\,\texttt{e}\,g \quad x^\dagger = \lambda k.\lambda g.k\,x\,g \quad \texttt{fun}(f,x,M)^\dagger = \lambda k.\lambda g.k\,\texttt{fun}(f,x,M^\dagger)\,g$$
$$(M_1\,M_2)^\dagger = \lambda k.\lambda g.{M_1}^\dagger\ (\lambda f.{M_2}^\dagger\lambda x.f\,x\,k)\,g$$
$$(M_1 \Box M_2)^\dagger = \lambda k.\lambda g.\texttt{br}\ ({M_1}^\dagger\,k\,g)\ ({M_2}^\dagger\,k\,g) \quad a^\dagger = \lambda k.\lambda g.a\ (\texttt{br}\ (k\,\texttt{e}\,g)\ (g\,(k\ \texttt{e})))$$

Here, the non-terminal *Sched* is defined by the rule $\mathit{Sched}\ x\ y \to x\ (\mathit{Sched}\ y)$, which schedules x first, passing to it the global continuation $\mathit{Sched}\ y$ (which will schedule y next). The terminal symbol `br` represents a non-deterministic branch. On the righthand side of the last translation rule, a and `e` are terminal symbols of arity 1 and 0 respectively. The program $M_1 \,||\, M_2$ is translated to a tree-generating program, which either schedules M_1 then M_2, or M_2 then M_1. Apart from the global action (the last rule), the translation of a thread is essentially the standard call-by-value CPS transformation except that a global continuation is passed as an additional parameter. The global action a is transformed to a tree node a, followed by a non-deterministic branch (expressed by `br`). The first branch evaluates the local continuation, while the second branch yields the control to the other thread by invoking the global continuation g.

By the definition of the transformation above, it should be clear that (i) if P is simply-typed, then $P^\dagger$ is a well-typed μHORS, and (ii) P has a sequence of global actions $a_1 a_2 \cdots a_n$ if and only if the tree generated by $P^\dagger$ has a path labeled with $a_1 a_2 \cdots a_n$ (with `br` ignored). Thus, verification of multi-threaded programs has been reduced to μHORS model checking; see [10] for more details.

Context-bounded model checking Qadeer and Rehof [19] showed that model checking of concurrent pushdown systems (or multi-threaded programs with *first-order* recursion) is decidable *if the number of context switches is bounded by a constant.* Our translation given above yields a generalization of the result: context-bounded model checking of multi-threaded, *higher-order* recursive programs is decidable. To obtain the result, it suffices to replace the scheduler *Sched* with Sched_ℓ given below, which allows only ℓ context switches:

$$\mathit{Sched}_0\ x\ y \to \texttt{e} \qquad \mathit{Sched}_{i+1}\ x\ y \to x\ (\mathit{Sched}_i\ y)$$

Then *Sched*$_i$'s have the following *non-recursive* types:

$$Sched_{2m} : \sigma_m \rightarrow \sigma_m \rightarrow \mathtt{o} \qquad Sched_{2m+1} : \sigma_{m+1} \rightarrow \sigma_m \rightarrow \mathtt{o}$$

where $\sigma_0 = \top$ and $\sigma_{i+1} = (\sigma_i \rightarrow \mathtt{o}) \rightarrow \mathtt{o}$. Thus, for the modified encoding $P^{\dagger_\ell}$, we have: (i) If P is simply-typed, then $P^{\dagger_\ell}$ is a well-typed μHORS *without recursive types*, and (ii) P with context-bound ℓ has a sequence of global actions $a_1 a_2 \cdots a_n$ if and only if the tree generated by $P^{\dagger_\ell}$ has a path labeled with $a_1 a_2 \cdots a_n$ (with `br` ignored). As an immediate corollary of the above properties and the decidability of HORS model checking [15], we obtain that context-bounded model checking of multi-threaded higher-order programs is decidable.[5]

5 Implementation and Experiments

We have implemented a prototype model checker RTRecS for μHORS based on the procedure FindCert described in Section 3. As the underlying SAT solver, we have used MiniSat 2.2 (`http://minisat.se/MiniSat.html`). We have also implemented a translator from FJ programs to μHORS based on Section 4.1.

The implementation is based on the procedure FindCert in Figure 2, except for the following points. RTRecS first performs an equality-based flow analysis [17] before model checking, and uses it as the equivalence relation $\sim$. $\mathcal{U}$ is also over-approximated by using the result of the flow analysis; thus, in the current implementation, the reductions of terms (3rd line in Figure 2) are performed only for finding a counter-example, without using the rule R-Eq.

Table 1 summarizes the result of preliminary experiments. (For space restriction, we omit some results, which are found in [10].) The programs used for the experiments and the web interface for our prototype are available at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/fjmc/`. The columns "#lines" and "#rules" show the number of lines of the source FJ program (if applicable) and the number of the rules of μHORS. The column "#states" shows the numer of states of the property automaton. The column "k" shows k of $\Gamma' \sqsubseteq_{\mathtt{s}} \Gamma^{(k)}$ in Section 3.2. The column "#sat" shows the number of sat clauses (i.e., the number of disjunctive formulas in conjunctive normal form) for the final call of the SAT solver. The column "time" shows the running time (excluding the time for translation from FJ to μHORS, which is anyway quite small).

$\mathcal{G}_1$ and $\mathcal{G}_2$ are from Examples 2 and 3, where the checked property is expressed by $\mathcal{B}_1$ in Example 4. `Thread` is the μHORS obtained from Example 12. The other programs were obtained from FJ programs, based on the translation discussed in Section 4.1, and except for `Twofiles`, we verified that the programs do not fail (where the meaning of "failure" depends on each program, as explained below). `Pred` is the running example in Section 4. The next six are list-manipulating programs (implemented as objects), which are small but

[5] We have considered only programs with two threads, but this restriction can be easily relaxed by using the same technique as Qadeer and Rehof [19].

programs	#lines	#rules	#states	k	#sat	time
$\mathcal{G}_1$	–	2	2	1	27	0.001
$\mathcal{G}_2$	–	3	2	1	49	0.002
`Thread`	–	9	5	1	38,171	0.580
`Pred`	21	15	1	1	157	0.005
`L-append`	20	30	1	1	165	0.006
`L-map`	43	182	1	1	738	0.235
`L-app-map`	43	212	1	1	1,546	0.391
`L-even`	25	87	1	1	249	0.025
`L-filter`	59	122	1	2	5,964	0.491
`L-risers`	73	64	1	2	17,419	0.445
`Twofiles`	28	21	5	2	739,867	13.86

Table 1. Experimental Results (CPU: Intel(R) Xeon(R) 3GHz, Memory: 8GB). Times are in seconds.

non-trivial programs. (In fact, `L-filter` and `L-risers` are object-oriented versions of benchmark programs of the PMRS verification tool [16].) For example, `L-filter` creates a list of natural numbers in a non-deterministic manner, filters out 0, and checks that the resulting list consists only of non-zero elements (and fails if it does not hold). See [10] for more details.

`Twofiles` was prepared as an example of verification of temporal properties. It is an object-oriented version of the program that accesses two files: one for read-only, and the other for write-only [9]. We verify that the read-only (write-only, resp.) file is closed after some reads (writes, resp.).

Our model checker RTRecS could successfully verify all the programs. The verification time and the size of SAT formulas were significantly larger for `Twofiles` compared with other programs. The explosion of the size of SAT formulas for `Twofiles` is due to the size of the automaton for describing the temporal property, which blows up the number of candidates of types to be considered. More optimizations are necessary for avoiding this problem. The number k was surprisingly small for all the benchmark programs; this indicates that our choice of $\sim$ based on the equality-based flow analysis provided a good approximation of types. Overall, the experimental results above are encouraging; we are not aware of other fully-automated (i.e. requiring no annotations), sound (i.e. no false negatives) verification tools that can verify all the programs above.

6 Related Work

The model checking of HORS has recently emerged as a new technique for verification of higher-order programs [15, 9, 14, 16, 13]. Except Tsukada and Kobayashi's work [24], however, all the previous studies dealt with *simply-typed* recursion schemes, which are not suitable for modeling objects. Tsukada and Kobayashi [24] studied model checking of *untyped* HORS and reduced it to a type checking problem for an infinite intersection type system. The latter problem is however undecidable and they did not provide any realistic procedure for model checking.

Several methods for model-checking functional programs have been proposed recently [21, 9, 14, 16, 25], and some of them [21, 14, 16] support recursive data structures (like lists). However, it is not clear how to extend them to support general recursive types (including negative occurrences of recursive type variables). Furthermore, many of them require annotations [21, 25] and are less precise.

There are previous studies on model checking of object-oriented programs [3, 4]. To our knowledge, however, they are based on finite state model checking; Java programs are either (i) abstracted to finite state models and then finite state model checkers are used to verify the abstract models, or (ii) directly model checked, but with an incomplete state exploration. In the former case, because of the huge semantic gap between object-oriented programs and finite state systems, a lot of information is lost by the translation from Java programs to models. In the latter case, a "model checker" is used mainly as a bug detection tool, instead of a verification tool. In contrast, our method uses μHORS as models, which are as expressive as source programs. No information is lost by the translation from FJ to μHORS, and no false alarms can be generated (although the model checker may not terminate for some valid programs). There are also other methods for verification or static analysis of object-oriented programs [1, 18, 23]. In general, they either require human intervention [1] or are fully automated but less precise than model checking. See [10] for more detailed discussion. Rowe and Bakel [22] proposed an intersection type system for reasoning about object-oriented programs, but did not give an automated verification algorithm.

There are many studies on model checking of recursive parallel programs [19, 6], which obtain decidable fragments by restricting synchronization primitives or applying approximations. It is interesting to see whether each result can be extended to higher-order, recursive parallel programs (besides context-bounded model checking discussed in Section 4.2).

Acknowledgments We thank Noriaki Nakano and anonymous reviewers for useful comments. This work is partially supported by Kakenhi 23220001.

References

1. M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The spec# programming system: Challenges and directions. In *Proceedings of VSTTE 2005*, volume 4171 of *LNCS*, pages 144–152. Springer-Verlag, 2005.
2. L. Cardelli. A semantics of multiple inheritance. *Info. Comput.*, 76(2/3):138–164, 1988.
3. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.
4. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000.

5. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23(3):396–450, 2001.
6. V. Kahlon. Reasoning about threads with bounded lock chains. In *Proceedings of CONCUR 2011*, volume 6901 of *LNCS*, pages 450–465. Springer-Verlag, 2011.
7. S. N. Kamin and U. S. Reddy. Two semantic models of object-oriented languages. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, chapter 13, pages 463–496. The MIT Press, 1993.
8. N. Kobayashi. Model-checking higher-order functions. In *Proceedings of PPDP 2009*, pages 25–36. ACM Press, 2009.
9. N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proc. of POPL*, pages 416–428, 2009.
10. N. Kobayashi and A. Igarashi. Model-checking higher-order programs with recursive types. An extended version available from `http://www-kb.is.s.u-tokyo.ac.jp/~koba/fjmc/`, 2012.
11. N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of LICS 2009*, pages 179–188, 2009.
12. N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science*, 7(4), 2011.
13. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and cegar for higher-order model checking. In *Proc. of PLDI*, 2011.
14. N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proc. of POPL*, pages 495–508, 2010.
15. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS 2006*, pages 81–90, 2006.
16. C.-H. L. Ong and S. Ramsay. Verifying higher-order programs with pattern-matching algebraic data types. In *Proc. of POPL*, pages 587–598, 2011.
17. J. Palsberg. Equality-based flow analysis versus recursive types. *ACM Trans. Prog. Lang. Syst.*, 20(6):1251–1264, 1998.
18. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proc. of POPL*, pages 75–86, 2008.
19. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proceedings of TACAS 2005*, volume 3440 of *LNCS*, pages 93–107. Springer-Verlag, 2005.
20. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Prog. Lang. Syst.*, 22(2):416–430, 2000.
21. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI 2008*, pages 159–169, 2008.
22. R. N. S. Rowe and S. van Bakel. Approximation semantics and expressive predicate assignment for object-oriented programming - (extended abstract). In *Proceedings of TLCA 2011*, volume 6690 of *LNCS*, pages 229–244, 2011.
23. C. Skalka. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation*, 21(3):239–282, 2008.
24. T. Tsukada and N. Kobayashi. Untyped recursion schemes and infinite intersection types. In *Proceedings of FOSSACS 2010*, volume 6014 of *LNCS*, pages 343–357. Springer-Verlag, 2010.
25. H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing programs via higher-order model checking. In *Proceedings of APLAS 2010*, volume 6461 of *LNCS*, pages 312–327. Springer-Verlag, 2010.