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) [33] has been recently studied as a new technique for automated verification of higher-order functional programs [22, 30, 34, 29]. 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 [22]; 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 [22, 30, 34, 45]. By combining it with predicate abstraction, a software model checker for functional programs can be constructed [34, 29]. The reason why HORS model checking works well for verification of functional programs is that HORS is itself a kind of functional program, so that the control structures (higher-order functions and recursion, in particular) of functional programs can be naturally and precisely modeled.

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) [15] (with only objects as primitive data) is Turing complete [41]. As for concurrency, the model checking of concurrent pushdown systems [38] 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) [22].

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 [22, 27, 46], 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, as shown in Figure 1. 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. For example, given an object-oriented program with a specification on method call sequences like "after a close method of a file object is called, no read/write methods are called"), we first construct a μHORS that generates a tree describing all the possible method call sequences. This is achieved by combining functional encoding of objects [6] with the reduction from verification of functional programs to HO model checking [22]. We then invoke the μHORS model checker to check whether the tree contains only valid method call sequences. The μHORS model checking can also be used to verify concurrent, higher-order programs, since we can write a scheduler in μHORS thanks to recursive types. The idea of applying program transformation to reduce program verification problems to HORS model checking problems is the same as our previous framework for automated verification of functional programs [22], but thanks to the expressive power of μHORS, we can now deal with other programming language features like objects and concurrency.

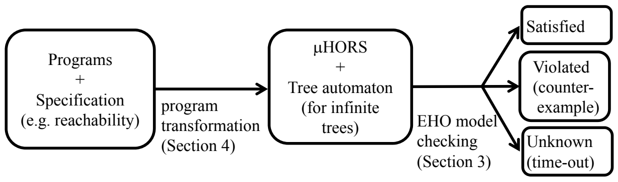


Fig. 1. The overall structure of our verification method

As a proof of concept, we have implemented a prototype of the μHORS model checker (which corresponds to the second phase of Figure 1), and a translator from Featherweight Java (FJ) programs [15] to μHORS (which corresponds to the first phase of Figure 1). Preliminary experiments show that we can indeed use the μHORS model checker to verify small but non-trivial object-oriented programs.

The rest of this paper is structured as follows. Section 2 introduces recursive intersection types and the μHORS model checking problem, and gives a recursive intersection type system that is sound with respect to the model checking problem. Section 3 gives an algorithm for μHORS model checking, and proves that it is sound and relatively complete with respect to the recursive intersection type system. Section 4 shows applications of μHORS model checking to verification of objects and multi-threads. Section 5 reports preliminary experiments. Section 6 discusses related work and limitations of our approach. Section 7 concludes the paper.

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 $\bigwedge_{i \in \{1,\dots,k\}} \alpha_i$ for $\bigwedge \{\alpha_1, \dots, \alpha_k\}$ 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 $\sigma_1, \dots, \sigma_m$, and return a value of type q. The type $\alpha_1 \wedge \cdots \wedge \alpha_k$ describes values that have all of the types $\alpha_1, \dots, \alpha_k$. 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 $\sigma_1, \dots, \sigma_m, \sigma'_1, \dots, \sigma'_m, q$ such that $E(\alpha_0) = \sigma_1 \to \cdots \to \sigma_m \to q$ and $E(\alpha'_0) = \sigma'_1 \to \cdots \to \sigma'_m \to q$, with $(\sigma'_1, \sigma_1), \dots, (\sigma'_m, \sigma_m) \in \mathcal{R}^\wedge$. Here, $\mathcal{R}^\wedge$ is:*
$\{(\bigwedge\{\alpha'_1, \dots, \alpha'_{k'}\}, \bigwedge\{\alpha_1, \dots, \alpha_k\}) \mid \forall i \in \{1, \dots, k\}.\exists j \in \{1, \dots, k'\}.\alpha'_j \mathcal{R} \alpha_i\}$.
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 [46], and the reduction from μHORS model checking to type checking (Theorem 2) is a corollary of the result of [46]. We shall therefore quickly go through the definitions and results; more formal definitions (apart from recursive types) and intuitions are found in [33, 22, 46].

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 $\{\mathsf{o}\}$ (where 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:

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

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 $\mathcal{N}, x_1{:}\kappa_1, \ldots, x_\ell{:}\kappa_\ell \vdash t : \mathtt{o}$.

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 [33] where the reduction relation $\longrightarrow$ is defined by: (i) $F\, t_1 \cdots t_\ell \longrightarrow [t_1/x_1, \ldots, t_\ell/x_\ell]t$ 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 $i \in \{1, \ldots, \ell\}$. See [33] 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 $[s_1/x_1, \ldots, s_\ell/x_\ell]t$ (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 $\{F_1\, x_1 \cdots x_{\ell_1} \to t_1, \ldots, F_m\, x_1 \cdots x_{\ell_m} \to t_m\}$ if $\mathcal{R}(F_i) = \lambda x_1. \cdots \lambda x_{\ell_i}.t_i$ for each $i \in \{1, \ldots, m\}$.*

Example 2. Consider μHORS $\mathcal{G}_1 = (\mathcal{N}_1, \Sigma_1, \mathcal{R}_1, S)$ where

$$\begin{aligned}
\mathcal{N}_1 &= \{S \mapsto \mathtt{o}, F \mapsto (\mathtt{o} \to \mathtt{o})\} \\
\Sigma_1 &= \{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0\} \\
\mathcal{R}_1 &= \{S \to F\, \mathtt{c}, \quad F\, x \to \mathtt{a}\, x\, (F\, (\mathtt{b}\, x))\}
\end{aligned}$$

S is rewritten as follows, and the tree in Figure 2 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:

$$\begin{aligned}
\mathcal{N}_2 &= \{S \mapsto \mathtt{o}, F \mapsto (\mathtt{o} \to \mathtt{o}), G \mapsto \mu\alpha.(\alpha \to \mathtt{o} \to \mathtt{o})\} \\
\mathcal{R}_2 &= \{S \to F\, \mathtt{c},\ F\, x \to G\, G\, x,\ G\, g\, x \to \mathtt{a}\, x\, (g\, g\, (\mathtt{b}\, x))\}
\end{aligned}$$

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.

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

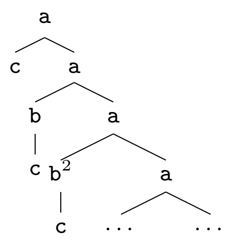


Fig. 2. The tree generated by $\mathcal{G}_1$ of Example 2.

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

$$\mathcal{N}_3 = \{S \mapsto \mathtt{o}, F \mapsto \mu\alpha.(\alpha \to \mathtt{o})\}$$
$$\mathcal{R}_3 = \{S \to F\,F,\ F\,x \to x\,x\},$$

Tree$(\mathcal{G}_3)$ is a singleton tree $\perp$. □

As usual [33, 22], 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 $dom(\Sigma)$, such that (i) its domain $dom(T)$ is non-empty and closed under the prefix operation, and (ii) if $\pi \in dom(T)$ then $\{j \mid \pi j \in dom(T)\} = \{1, \ldots, \Sigma(T(\pi))\}$. A (deterministic) *trivial automaton* [1] $\mathcal{B}$ 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 $Q \times dom(\Sigma)$ 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) $dom(T) = dom(R)$; (ii) $R(\epsilon) = q_0$; and (iii) for every $\pi \in dom(R)$, $\delta(R(\pi), T(\pi)) = R(\pi 1) \cdots R(\pi \Sigma(T(\pi)))$. For a trivial automaton $\mathcal{B} = (\Sigma, Q, \delta, q_0)$ (with $\perp \notin dom(\Sigma)$), 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 [28] 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:

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

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 2. □

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 $\mathcal{B}$.[3] The problem is in general undecidable.

Theorem 1 ([46]). *The model checking problem for μHORS is undecidable.*

[46] gives only a proof sketch. An alternative proof can be obtained by using the reduction from FJ [15] (which is Turing-complete) to μHORS given in Section 4.1.

Type system for model checking 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 [22].

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

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

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

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

The following theorem is a special case of the soundness of Tsukada and Kobayashi's infinite intersection type system for untyped HORS [46].

Theorem 2 (soundness). *Let $\mathcal{B}$ be a trivial automaton $(\mathit{\Sigma}, Q, \delta, q_{\mathcal{B},0})$ and $\mathcal{G}$ be a μHORS. If $\mathit{\Gamma} \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$, then $\mathit{Tree}(\mathcal{G})$ is accepted by $\mathcal{B}^{\perp}$.*

Remark 2. If $\mathcal{G}$ does not use recursive sorts, the type system is also complete [22]. The model checking algorithm discussed in this paper is, therefore, complete for the class of μHORS without recursive sorts.

Example 5. Recall $\mathcal{G}_1$ and $\mathcal{G}_2$ in Examples 2 and 3, and $\mathcal{B}_1$ in Example 4. $\mathit{\Gamma}_1 \vdash_{\mathcal{B}_1} (\mathcal{G}_1, S) : q_0$ and $\mathit{\Gamma}_2 \vdash_{\mathcal{B}_1} (\mathcal{G}_2, S) : q_0$ hold for:

$$\begin{array}{l}\mathit{\Gamma}_1 = \{S : q_0, F : (q_0 \wedge q_1) \to q_0\} \\ \mathit{\Gamma}_2 = \mathit{\Gamma}_1 \cup \{G : \mu\alpha.(\alpha \to (q_0 \wedge q_1) \to q_0)\} \quad \Box\end{array}$$

Given a type environment $\mathit{\Gamma}$, a μHORS $\mathcal{G}$, and an automaton $\mathcal{B}$, it is decidable whether $\mathit{\Gamma} \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ holds. Thus, $\mathit{\Gamma}$ can be used as a certificate for

[3] We consider $\mathcal{B}^{\perp}$ instead of $\mathcal{B}$ as we are interested in safety properties. $\mathcal{G}_3$ in Remark 1 generates no nodes, hence never violates safety properties; indeed, $\mathit{Tree}(\mathcal{G}_3)$ is accepted by $\mathcal{B}^{\perp}$ for any $\mathcal{B}$.

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. To establish the converse, we need to replace (finite) recursive intersection types with infinite intersection types [46]. We have the following properties on the (un)decidability of type checking.

Theorem 3. *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. More precisely, the set $\{\mathcal{G} \mid \exists \Gamma.\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}\}$ is recursively enumerable but not recursive.

Proof. The first property follows from the fact that the equality and subtyping relations on (finite) recursive intersection types are decidable. To see that $\{\mathcal{G} \mid \exists \Gamma.\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}\}$ is recursively enumerable, it suffices to observe that for each $\mathcal{G}$, we can enumerate all the possible candidates of Γ and check whether $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ holds. The set not being recursive follows from the fact that one can encode a Minsky machine [32] M into a μHORS $\mathcal{G}_M$, so that M halts if and only if $\mathcal{G}_M$ is typable in the recursive intersection type system. Let $\mathcal{B}$ be an automaton that accepts the singleton language $\{\texttt{end}\}$. Given a Minsky machine M, prepare the following non-terminal symbols:

- The start symbol S, of sort $\mathtt{o}$.
- $\mathit{Zero} : \mathbf{N}$, $\mathit{Succ} : \mathbf{N} \to \mathbf{N}$, $\mathit{Pred} : \mathbf{N} \to \mathbf{N}$, $\mathit{Eq} : \mathbf{N} \to \mathbf{N} \to \mathtt{o} \to \mathtt{o} \to \mathtt{o}$ that simulates constructors and operations on natural numbers. Here, $\mathbf{N}$ is a certain recursive type of terms obtained by encoding natural numbers.
- F_i of sort $\mathbf{N} \to \mathbf{N} \to \mathbf{N} \to \mathbf{N} \to \mathtt{o}$, for each program counter i.

The first and second arguments of F_i are the values of two counters. The third and fourth arguments keep the number of execution steps and they are expected to be identical. We prepare the following rules:

- $S \to F_0\ \mathit{Zero}\ \mathit{Zero}\ \mathit{Zero}\ \mathit{Zero}$ (assuming that 0 is the initial program counter).
- Appropriate rules for *Zero*, *Succ*, *Pred*, and *Eq* that simulate the corresponding constructors and operations on natural numbers (this may involve additional non-terminals and rules on them). In particular, the following properties should hold.

  $$\mathit{Eq}\ t_1\ t_2\ t_3\ t_4 \longrightarrow^*_{\mathcal{G}} t_3 \text{ if } [\![t_1]\!] = [\![t_2]\!] = n \text{ for some natural number } n$$
  $$\mathit{Eq}\ t_1\ t_2\ t_3\ t_4 \longrightarrow^*_{\mathcal{G}} t_4 \text{ if } [\![t_1]\!] = m \neq [\![t_2]\!] = n \text{ for some natural numbers } m, n$$

  Here, $[\![\,\cdot\,]\!]$ is a partial function from the set of terms to the set of natural numbers, defined by:

  $$[\![\mathit{Zero}]\!] = 0 \qquad [\![\,\mathit{Succ}(t)]\!] = [\![t\,]\!] + 1$$
  $$[\![\mathit{Pred}(t)]\!] = \begin{cases} 0 & \text{if } [\![\,t]\!] = 0 \\ n-1 & \text{if } [\![\,t]\!] = n > 0 \\ \text{undefined otherwise} \end{cases}$$

– For each program counter i, prepare a rule that simulates the instruction I_i after checking the equivalence of the third and fourth arguments of F_i. So,
  - If I_i is "`halt`," then
$$F_i\ c_0\ c_1\ x\ y \to \mathit{Eq}\ x\ y\ \texttt{end}\ \texttt{fail}$$
  - If I_i is "$c_j := c_j + 1; \texttt{goto}\ \ k$," then
$$F_i\ c_0\ c_1\ x\ y \to \mathit{Eq}\ x\ y\ (F_k\ t_0\ t_1\ (\mathit{Succ}\ x)\ (\mathit{Succ}\ y))\ \texttt{fail},$$
    where $t_{j'} = \mathit{Succ}\ c_j$ if $j' = j$ and $t_{j'} = c_{j'}$ otherwise, for $j' \in \{0, 1\}$.
  - If I_i is "$\texttt{if}\ c_i = 0\ \texttt{then}\ \texttt{goto}\ \ k_1\ \texttt{else}\ \ c_i := c_i - 1; \texttt{goto}\ \ k_2$," then
$$\begin{array}{l} F_i\ c_0\ c_1\ x\ y \to \mathit{Eq}\ x\ y \\ \qquad\qquad\qquad (\mathit{Eq}\ \mathit{Zero}\ c_i\ (F_{k_1}\ c_0\ c_1\ (\mathit{Succ}\ x)\ (\mathit{Succ}\ y)) \\ \qquad\qquad\qquad\qquad\qquad (F_{k_2}\ t_0\ t_1\ (\mathit{Succ}\ x)\ (\mathit{Succ}\ y))\ ) \\ \qquad\qquad\qquad \texttt{fail} \end{array}$$
    where $t_{j'} = \mathit{Pred}\ c_j$ if $j' = j$ and $t_{j'} = c_{j'}$ otherwise, for $j' \in \{0, 1\}$.

Let $\mathcal{G}_M$ be the resulting grammar. We claim that M halts if and only if $\mathcal{G}_M$ is typable in the recursive intersection type system. If M halts, then there is a finite reduction sequence $S \longrightarrow^*_{\mathcal{G}_M} \texttt{end}$. By the result on ordinary higher-order model checking [24], $\mathcal{G}_M$ is typable in the intersection type system even without recursive types. Suppose M does not halt, but $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ for a finite, recursive type environment Γ. Then, the set U of applicative terms that have type $q_{\mathcal{B},0}$ under Γ must be regular (note that each typing rule can be viewed as a transition rule for an alternating tree automaton that accept term trees, having intersection types as states). Furthermore, by the subject reduction property and the condition $S \in U$, (i) U contains all the terms obtained by reducing S; and (ii) any element of U cannot be reduced to `fail`. By condition (ii), if $F_i\ t_1\ t_2\ (\mathit{Succ}^m\ \mathit{Zero})\ (\mathit{Succ}^n\ \mathit{Zero}) \in U$, then it must be the case that $m = n$. By condition (i) and the fact that M does not halt, for every n, there exists i, t_1, t_2 such that $F_i\ t_1\ t_2\ (\mathit{Succ}^n\ \mathit{Zero})\ (\mathit{Succ}^n\ \mathit{Zero}) \in U$. But then U cannot be regular, hence a contradiction. $\square$

3 Model Checking μHORS

We now describe the main result of this paper: a model checking procedure for μHORS (the second phase in Figure 1).[4] We shall develop a (possibly non-terminating) procedure Check that satisfies:

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

[4] Readers who are curious about the relationship between program verification and μHORS may wish to consult Section 4 and then come back to this section.

By Theorem 3, the procedure Check can only be a semi-algorithm: it may not terminate if $\mathit{Tree}(\mathcal{G})$ 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 [20], 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 [20], 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) [20]. 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 4 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 5 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 $\mathtt{o}$) 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:

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

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. The figure shows a state obtained by a finite number of reductions. From an infinite fair reduction sequence, we obtain the following set as $\mathcal{U}$:

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

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 [20]. 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 [20] because of the presence of recursive types and the term equivalence relation $\sim$.

Let $(\mathcal{X}_0,\mathcal{U}_0) \longrightarrow_\sim (\mathcal{X}_1,\mathcal{U}_1) \longrightarrow_\sim (\mathcal{X}_2,\mathcal{U}_2) \longrightarrow_\sim \cdots$ 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.

By the definition above, we have:

Lemma 1. *If the number of equivalence classes induced by $\sim$ is finite, then $\Gamma_{\mathcal{X},\mathcal{U},\sim}$ is also finite. Furthermore, there exists m such that $\Gamma_{\mathcal{X}_m,\mathcal{U}_m,\sim} = \Gamma_{\mathcal{X},\mathcal{U},\sim}$.*

Although $\Gamma_{\mathcal{X},\mathcal{U},\sim}$ above is defined for a fair reduction sequence that is usually *infinite*, Lemma 1 above says that $\Gamma_{\mathcal{X},\mathcal{U},\sim}$ can be computed from a *finite* prefix of the fair reduction sequence.

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

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

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

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

Even if $\mathcal{G}$ is typable, $\Gamma_{\mathcal{X},\mathcal{U},\sim}$ may not be a proper witness, i.e., $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ may not hold in general. The theorem below (Theorem 4), however, ensures that if $\mathcal{G}$ is typable and if $\sim$ is properly chosen, $\Gamma_{\mathcal{X},\mathcal{U},\sim}$ is indeed 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 4. *If $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ and $\sim \subseteq \sim_\Gamma$, then $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$.*

Proof. See Appendix A.1.

Example 8. Recall $\mathcal{G}_2$ in Example 3, and $\Gamma_2 = \{S : q_0, F : (q_0 \land q_1) \to q_0, G : \mu\alpha.(\alpha \to (q_0 \land q_1) \to q_0)\}$ in Example 5. $\sim_{\Gamma_2}$ is a congruence that satisfies $\mathtt{bc} \sim_{\Gamma_2} \mathtt{c}$, $S \sim_{\Gamma_2} F\,\mathtt{c}$, and $F \sim_{\Gamma_2} G\,G$. The relation $\sim$ in Example 6 satisfies the assumption $\sim \subseteq \sim_{\Gamma_2}$ of Theorem 4, and $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}_1} (\mathcal{G}_2, S) : q_0$ holds indeed.

Theorem 4 cannot be directly used for type inference, since we do not know $\sim_\Gamma$ in advance. We shall prove below (in Theorem 5) 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 $\Gamma_1 \sqsubseteq \Gamma_2$ if $dom(\Gamma_1) \subseteq dom(\Gamma_2)$ and for every $x : \tau_1 \in \Gamma_1$, there exists τ_2 such that $x : \tau_2 \in \Gamma_2$ and $\tau_1 \sqsubseteq \tau_2$.

Example 9. Let τ_1 be $q_1 \to q_2$ and τ_2 be $(q_1 \land q_0) \to q_2$. 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 $\tau_1 \to q_0 \not\leq \tau_2 \to q_0$. For τ_0 and τ_1 in Example 1, $\tau_0 \sqsubseteq \tau_1$ but $\tau_1 \not\sqsubseteq \tau_0$. $\mathcal{R}_2 = \{(\tau_0, \tau_1), (\tau_0, \tau_2)\}$ is a witness for $\tau_0 \sqsubseteq \tau_1$. $\square$

Theorem 5. *Suppose $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$. 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 Γ' such that $\Gamma' \sqsubseteq \Gamma_{\mathcal{X},\mathcal{U},\sim}$ and $\Gamma' \vdash (\mathcal{G}, S) : q_{\mathcal{B},0}$,*

The proof is given in Appendix A. Intuitively, Theorem 5 holds because, if $\sim$ is not a subset of $\sim_\Gamma$, 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 the equivalence relation: $\sim^{(1)} \cup \{(G\,G, \mathtt{b}), (\mathtt{b}, G\,G)\}$. By using $\sim^{(2)}$, we get the reductions as shown in Figure 3. In the figure, the

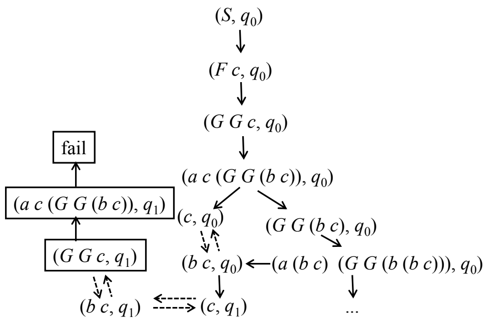


Fig. 3. Reductions based on $\sim^{(2)}$

extra nodes caused by using $\sim^{(2)}$ instead of $\sim^{(1)}$ are surrounded by boxes. From the reductions, we obtain the following type equations:

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

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

Theorem 5 yields the procedure FINDCERT in Figure 4. We assume that `expandEq`$(\sim, \mathcal{X}')$ in Figure 4 returns an equivalence relation $\sim'$ on $\mathcal{X}'$ such that $\sim \subseteq \sim'$ and $|\mathcal{X}'/\sim'| \le n$. for some fixed n. The condition $\exists \Gamma'.\Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0} \wedge \Gamma' \sqsubseteq \Gamma$ is in general undecidable because of the presence of recursive types. Thus, we bound the size (i.e., the number of type constructors) of Γ' by v, and gradually increase the bound. An algorithm to check whether there exists Γ' such that $|\Gamma'| < v$ and $\Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0} \wedge \Gamma' \sqsubseteq \Gamma$ is discussed in Section 3.2. By Theorem 5, we have:

Theorem 6 (relative completeness). *If $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ for some finite recursive type environment Γ, then* FINDCERT$(\mathcal{G}, S, q_{\mathcal{B},0})$ *eventually terminates and outputs Γ' such that $\Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$.*

The termination is ensured by Lemma 1.

3.2 Type Checking by SAT Solving

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

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

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:

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

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 $b_{\Delta, x_1:\xi_1, \ldots, x_k:\xi_k \vdash s:\rho}$, which expresses whether $f(\Delta, x_1 : \xi_1, \ldots, x_k : \xi_k) \vdash_{\mathcal{B}} s : f(\rho)$ should hold. (ii) For each pair (ρ_1, ρ_2) of types occurring in Δ, we introduce $b_{\rho_1 \leq \rho_2}$, which expresses whether $f(\rho_1) \leq f(\rho_2)$ 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 $F :^{b_1} \rho_1, \ldots, F :^{b_n} \rho_n \in \Delta$ 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) $b \Rightarrow b_{\Delta, x_1:\xi_1, \ldots, x_k:\xi_k \vdash t:q}$
for each $F :^{b} \xi_1 \to \cdots \to \xi_k \to q \in \Delta$ such that $\mathcal{R}(F) = \lambda x_1, \ldots, x_k.t$.

(iii) $b_{\Delta', x:\bigwedge_{j \in J} b_j \rho_j \vdash x:\rho} \Rightarrow \bigvee_{j \in J} (b_j \wedge b_{\rho_j \leq \rho})$, for each $b_{\Delta', x:\bigwedge_{j \in J} b_j \rho_j \vdash x:\rho}$.

(iv) $b_{\Delta' \vdash a:\rho} \Rightarrow \bigvee \{ b_{q_1 \to \cdots \to q_k \to q \leq \rho} \mid \delta(a, q) = q_1 \cdots q_k \}$, for each $b_{\Delta' \vdash a:\rho}$.

(v) $b_{\Delta' \vdash t_1 t_2:\rho} \Rightarrow \bigvee (b_{\Delta' \vdash t_1:(\bigwedge_{j \in J} b_j \rho_j) \to \rho} \wedge (\bigwedge_{j \in J} (b_j \Rightarrow \bigvee (b_{\Delta' \vdash t_2:\rho'} \wedge b_{\rho' \leq \rho_j}))))$,
for each $b_{\Delta' \vdash t_1 t_2:\rho}$.

(vi) $b_{(\bigwedge_{i \in I} b_i \rho_i) \leq (\bigwedge_{j \in J} b_j \rho'_j)} \Rightarrow \bigwedge_{j \in J} (b_j \Rightarrow \bigvee_{i \in I} (b_i \wedge b_{\rho_i \leq \rho'_j}))$,
for each $b_{(\bigwedge_{i \in I} b_i \rho_i) \leq (\bigwedge_{j \in J} b_j \rho'_j)}$.

(vii) $b_{\xi_1 \to \cdots \to \xi_k \to q \leq \xi'_1 \to \cdots \to \xi'_m \to q'} \Rightarrow k = m \wedge q = q' \wedge \bigwedge_{i \in \{1, \ldots, k\}} b_{\xi'_i \leq \xi_i}$,
for each $b_{\xi_1 \to \cdots \to \xi_k \to q \leq \xi'_1 \to \cdots \to \xi'_m \to q'}$.

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 $f(\Delta, x_1{:}\xi_1, \ldots, x_k{:}\xi_k) \vdash_{\mathcal{B}} t : f(\rho)$, 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:

$$\Gamma_{\mathcal{X}^{(2)}, \mathcal{U}^{(2)}, \sim^{(2)}} = \{ S : q_0, F : (q_0 \wedge q_1) \to q_0, G : \tau_0, G : \tau_1 \},$$

where $\tau_i = (\tau_1 \wedge \tau_2) \to (q_0 \wedge q_1) \to q_i$ for $i \in \{0, 1\}$. To find a type environment Γ such that $\Gamma \sqsubseteq \Gamma_{\mathcal{X}^{(2)}, \mathcal{U}^{(2)}, \sim^{(2)}}$ and $\Gamma \vdash_{\mathcal{B}_1} (\mathcal{G}_2, S) : q_0$, let us add boolean variables to type bindings and types as follows:

$$\begin{array}{l} \Delta = \{ S : q_0, F : (q_0 \wedge q_1) \to q_0, G :^{b_0} \alpha_0, G :^{b_1} \alpha_1) \} \\ \alpha_i = (b_2 \alpha_0 \wedge b_3 \alpha_1) \to (q_0 \wedge q_1) \to q_i \quad (i \in \{0, 1\}) \end{array}$$

(Here, for the sake of simplicity, we have added boolean variables only to critical parts, assuming that the values of other boolean variables are true; we also apply various simplifications below.) From the typing of G, we get the following boolean constraints:

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

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 $f(b_0) = f(b_2) = \texttt{true}$ and $f(b_1) = f(b_3) = \texttt{false}$. Thus, we get

$$\begin{aligned} &\Gamma' = f(\Delta) = \{S : q_0, F : (q_0 \wedge q_1) \to q_0, G : \tau_0\} \\ &\tau_0 = \tau_0 \to (q_0 \wedge q_1) \to q_0. \end{aligned}$$

We have $\Gamma' \vdash_{\mathcal{B}_1} (\mathcal{G}_2, S) : q_0$ as required. □

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 programswritten in (a call-by-value variant of) Featherweight Java (FJ) [15] to μHORS model checking problems. The idea is to transform an FJ source program into μHORS that generates a tree representing all the possible action sequences 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, that is, whether program execution reaches certain program points;
– order of method invocations; and
– whether downcasts may fail.

After we introduce the source language FJ-- briefly, we give a formal translation from FJ-- to μHORS and discuss how the properties above can be model-checked.

FJ-- We first describe main differences of FJ-- from FJ. In FJ--, there is a construct to signal certain actions, which correspond to terminal symbols in μHORS. So, (multi-step) reduction relation is augmented with a sequence of actions that occur during reduction. We also add a non-deterministic choice operator, which will be necessary when we apply predicate abstractions [29].

To ease presentation, FJ-- assumes a few simplifications:

– Methods are called by value.
– Field access is allowed only in the form `this`.f.

– For each method name, its arity (the number of arguments) is globally fixed.

Not only is the language call-by-value, but also the order of evaluation of subexpressions is fixed. The second restriction is not essential because we can add getter methods and replace any field access to an invocation of a corresponding getter method. The third restriction is an artifact of our translation but does not weaken the expressive power of the language.

Now we briefly introduce FJ`--`. As usual, we assume class names, ranged over by $C, D, \ldots$, field names, ranged over by f, g, method names, ranged over by m. Let the set of action names, ranged over by a, be *Actions*. The syntax of FJ`--` is given as follows:

$$\begin{array}{lll}
L & ::= \texttt{class}\ C\ \texttt{extends}\ C\ \{\widetilde{C}\ \widetilde{f}; K\ \widetilde{M}\} & \text{(classes)} \\
K & ::= C(\widetilde{C}\ \widetilde{f})\{\texttt{super}(\widetilde{f}); \texttt{this}.\widetilde{f} = \widetilde{f};\} & \text{(constructors)} \\
M & ::= C\ m(\widetilde{C}\ \widetilde{x})\{s\} & \text{(methods)} \\
v & ::= x \mid \texttt{this}.f \mid \texttt{new}\ C(\widetilde{v}) & \text{(values)} \\
s & ::= \texttt{return}\ v; \mid C\ x = v_0.m(\widetilde{v}); s & \text{(statements)} \\
 & \mid\ \ a; s \mid s_1 \Box s_2 \mid \mathit{fail}; &
\end{array}$$

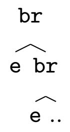
Unlike the original definition of FJ, a method body is required to be written in A-normal form [10] to make the definition of the translation more concise. The statement of the form $C\ x = v_0.m(\widetilde{v}); s$ first invokes method m on v_0, bind x to the result, and executes s. The statement of the form $a; s$ signals a and executes s. The statement $s_1 \Box s_2$ executes s_1 or s_2 in a non-deterministic manner. The statement $\mathit{fail};$ causes abnormal termination of the execution. We omit downcasts from FJ`--` but it is easy to add (see also Section 4.1). We denote the set of free variables in a statement s by $FV(s)$. An FJ`--` program is a pair (CT, s) of a class table (a mapping from class names to classes) and a statement, which corresponds to the body of the `main` method. We assume the same sanity conditions on the class table as in FJ.

The definitions of auxiliary functions $\mathit{fields}(C)$, $\mathit{mtype}(m, C)$, $\mathit{mbody}(m, C)$ to look up fields, method types, and method bodies are essentially the same as those of FJ and so omitted. The type system of FJ`--` is a straightforward adaptation of that of FJ and so we omit typing rules but write $\vdash (CT, s) : C$ to mean that CT is a well-formed class table and s is given type C under CT. Operational semantics is given by a multi-step reduction relation written $s \xrightarrow{a_1 \cdots a_n} r$, where r is either a statement s' or a special symbol $\bullet$, which means termination. The relation means that statement s is reduced to r in multiple steps and actions $a_1, \ldots, a_n$ occur during reduction in this order. When r is $\bullet$, the last action is either `e`, which stands for normal termination by `return`ing a value, or `fail` for abnormal termination caused by *fail* or `NoSuchMethodError`. We also omit reduction rules, which are also straightforward.

We use the following classes that represent natural numbers with methods for addition (`add`) and predecessors (`pred`) as a running example. Method `rand` non-deterministically returns a natural number that is equal to or greater than the argument. 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; □ return new S(this.rand(n));
  }
}
class Z extends Nat { // Zero
   Nat add(Nat n) { return n; }
}
class S extends Nat { // Successor
  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:



Here, `br` and `e` represent non-deterministic branch (caused by □) 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 [7], 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.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; }
```

in class C is represented by a non-terminal C_m, whose body is

$$\lambda x_1.\cdots\lambda x_n.\lambda this.\lambda k.\,[\![\, e]\!]_k$$

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 [18]. 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` (as allowed in FJ) 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:

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

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

A global action a 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`.

There are some further twists in giving encoding:

– 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 $\lambda\widetilde{x}.\lambda k.\texttt{fail}$ in the corresponding position of the tuple. The dummy function just outputs `fail` to signal `NoSuchMethodError` whenever it is called. For example, for our running example, if there is another class that defines a method *print*, a Z object would be expressed as a tuple $\langle Z_{add}, Z_{pred}, Z_{rand}, Z_{print}\rangle$ where $Z_{print} = \lambda this.\lambda k.\texttt{fail}$.

– As careful readers may recall, μHORS does not have tuples as primitives. Thus, we actually curry all the functions that take tuples as arguments. For example, the example of method invocation shown above is replaced with

$$S\_add\ Z\_add\ Z\_pred\ Z\_rand\ S\_add\ S\_pred\ S\_rand\ k$$

Thanks to the CPS representation, tuples occur only in the argument positions, so that currying transformation eliminates all the tuples. Thus, we can assume below, without loss of generality, that μHORS has been extended with tuples and tuple types.

It is easy to see that 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:

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

Now, we develop translation formally. Let ℓ be the number of defined method names and $Meth = \{m_1, \dots, m_\ell\}$ be the set of defined method names, i.e., $\{m \mid \exists C.mbody(m, C) = \widetilde{x}.s\}$, in a given class table. We write $arity(m)$ for the number of arguments to method m and $\langle \boldsymbol{x} \rangle$ for $\langle x_1, \dots, x_\ell \rangle$.

$$\begin{array}{ll}
[\![x]\!] & = \langle \boldsymbol{x} \rangle \\
[\![\mathtt{this}.f]\!] & = \langle \boldsymbol{f} \rangle \\
[\![\mathtt{new}\ C(\widetilde{v})]\!] & = \langle C\_m_1\ [\![\, v_1\,]\!] \cdots [\![\, v_n]\!], \dots, C\_m_\ell\ [\![\, v_1\,]\!] \cdots [\![\, v_n]\!] \rangle \\
 & \\
[\![\mathtt{return}\ v]\!]_k & = (k\ [\![\, v]\!], \emptyset) \\
[\![C\ x = v_0.m_i(\widetilde{v}); s]\!]_k & = (t_i\ [\![\, v_1\,]\!] \cdots [\![\, v_n\,]\!]\ [\![v_0\,]\!]\ \ (F\ \langle \boldsymbol{z_1} \rangle \cdots \langle \boldsymbol{z_m} \rangle\ k), \mathcal{R} \cup \{F \mapsto \lambda\langle \boldsymbol{z_1} \rangle \cdots \lambda\langle \boldsymbol{z_m} \rangle.\lambda k'.\lambda\langle \boldsymbol{x} \rangle.t\}) \\
 & \qquad\quad \text{where } \langle t_1, \dots, t_\ell \rangle = [\![v_0]\!], \\
 & \qquad\qquad (t, \mathcal{R}) = [\![s]\!]_{k'}, \{z_1, \dots, z_m\} = FV(s) \setminus \{x\} \text{ for fresh } F \text{ and } k', \\
[\![a; s]\!]_k & = (a\ (t\ k), \mathcal{R}) \qquad \text{where } (t, \mathcal{R}) = [\![s]\!]_k \\
[\![s_1 \Box s_2]\!]_k & = (\mathtt{br}\ (t_1\ k)\ (t_2\ k), \mathcal{R}_1 \cup \mathcal{R}_2) \qquad \text{where } (t_i, \mathcal{R}_i) = [\![s_i\,]\!]_k \text{ for } i = 1, 2 \\
[\![fail;]\!]_k & = (\mathtt{fail}, \emptyset)
\end{array}$$

Fig. 5. Translation of values and statements.

Translation of Values and Statements. We first show how values and statements are translated in Figure 5. A value is translated to a tuple of methods. A variable or field access is expressed by a tuple of variables of length ℓ. An object instantiation `new` $C(\widetilde{v})$ is expressed by a tuple of applications of methods to field values. As we have mentioned, C_m_i is a non-terminal that expresses the body of method m_i in class C. For example,

$$\begin{array}{l}[\![\mathtt{new\ S(p')}]\!] = \\ \langle S\_add\ \langle p'_a, p'_p, p'_r \rangle, S\_pred\ \langle p'_a, p'_p, p'_r \rangle, S\_rand\ \langle p'_a, p'_p, p'_r \rangle \rangle\ .\end{array}$$

(We use subscripts a, p and r to mean methods `add`, `pred` and `rand`, rather than numbers.)

A statement is translated to a pair $(t, \mathcal{R})$ of a λ-term and a set of rewriting rules. Translation is essentially call-by-value CPS-translation and the definitions of continuation functions are returned as the set of rewriting rules. A statement `return` v; is translated to an application of the given continuation to the value v; no continuation function is generated. In translation of a method invocation, t_i, the i-th element of v_0, represents the method to be invoked. It is applied to the formal arguments, the receiver object, and then the continuation F of the invocation. The body of F is obtained by translating s; since s may refer to other variables than x (the result of method invocation), F takes values of those free variables as arguments. So, the translation combines CPS translation and λ-lifting [16]. Translation of action signaling, non-deterministic choice, and failure is straightforward.

For example, given k, the body of `add` in class `S` is translated to

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

The invocation of `add` on `p` is expressed by application of p_a to terms that express the actual argument `n`, the receiver, and continuation F.

$$\begin{array}{l}[\![\texttt{class}\ C\ \texttt{extends}\ D\ \{C_1\ f_1; \cdots C_n\ f_n; K\ \widetilde{M}\}]\!] =\\ \quad (\bigcup\{\{C\_m_i \mapsto \lambda\langle \boldsymbol{g_1}\rangle. \cdots \lambda\langle \boldsymbol{g_k}\rangle.t\} \cup \mathcal{R} \mid\\ \qquad \mathit{fields}(C) = D_1\ g_1, \ldots, D_k\ g_k, C_0\ m_i(\widetilde{C}\ \widetilde{x})\{s\} \in \widetilde{M}, (t, \mathcal{R}) = [\![\widetilde{x}.s]\!]\})\\ \quad \cup\{C\_m_i \mapsto \lambda\langle \boldsymbol{f_1}\rangle. \cdots \lambda\langle \boldsymbol{f_n}\rangle.\lambda\langle \boldsymbol{g_1}\rangle. \cdots \lambda\langle \boldsymbol{g_k}\rangle.\lambda k.D\_m_i\ \langle \boldsymbol{g_1}\rangle \cdots \langle \boldsymbol{g_k}\rangle\ k \mid\\ \qquad m_i \notin \widetilde{M}, \mathit{fields}(D) = D_1\ g_1, \ldots, D_k\ g_k\}\\ [\![\widetilde{x}.s]\!] = (\lambda\langle \boldsymbol{x_1}\rangle. \cdots \lambda\langle \boldsymbol{x_n}\rangle.\lambda\langle \boldsymbol{this}\rangle.\lambda k.t, \mathcal{R}) \qquad \text{where } (t, \mathcal{R}) = [\![s\,]\!]_k \ \text{ for fresh } k\end{array}$$

$$\begin{array}{l}[\![(CT, s)]\!] = (\mathcal{N}, \Sigma, \mathcal{R}, S)\\ \quad \text{where}\, \Sigma = \{\texttt{br} \mapsto 2, \texttt{fail} \mapsto 0, \texttt{e} \mapsto 0\} \cup \{a \mapsto 1 \mid a \in \mathit{Actions}\}\\ \qquad\quad \mathcal{R} = \{\mathit{Object\_m_i} \mapsto \lambda\langle \boldsymbol{x_1}\rangle \cdots \lambda\langle \boldsymbol{x_{n+1}}\rangle.\lambda k.\texttt{fail} \mid m_i \in \mathit{Meth}, \mathit{arity}(m_i) = n\}\\ \qquad\qquad \cup \bigcup_C [\![CT(C)\,]\!] \cup \{S \mapsto t, \mathit{End} \mapsto \lambda\langle \boldsymbol{x}\rangle.\texttt{e}\} \cup \mathcal{R}'\\ \qquad\qquad\quad \text{where } (t, \mathcal{R}') = [\![s]\!]_{\mathit{End}}\\ \qquad\quad \mathcal{N} = \{F \mapsto \mathit{OType}(n) \mid \mathcal{R}(F) = \lambda\langle \boldsymbol{x_1}\rangle \cdots \lambda\langle \boldsymbol{x_n}\rangle.\lambda k.t\} \cup \{S \mapsto \texttt{o}, \mathit{End} \mapsto \kappa_o \to \texttt{o}\}\\ \qquad\qquad\quad \text{where} \begin{cases} \kappa_o & = \kappa_1 \times \cdots \times \kappa_\ell\\ \mathit{OType}(0) & = (\kappa_o \to \texttt{o}) \to \texttt{o}\\ \mathit{OType}(n+1) & = \kappa_o \to \mathit{OType}(n)\end{cases}\end{array}$$

Fig. 6. Translation of Classes and Programs.

Translation of Class Definitions and Programs. A class definition is translated to a set of rules in (the upper half of) Figure 6. For each class C, non-terminals $C\_m_1, \ldots, C\_m_\ell$ are generated. When m_i is defined in C, its body is translated to t (with a set $\mathcal{R}$ of continuation functions) and t is further parametrized by fields. Otherwise, C_m_i calls the inherited definition D_m_i. Since subclass C may have additional fields, C_m_i has to be further parametrized by those fields. As we will see below, $Object\_m_i$ is a function that just signals a failure. So, such functions will be inherited to subclasses until a subclass gives a definition of m_i.

For example, $[\![$`class Z` $\cdots]\!]$ includes

$$Z\_add \mapsto \lambda k.Nat\_add\ k$$

since `add` is inherited from the superclass and $[\![$`class S` $\cdots]\!]$ includes the following two rules:

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

Here, the formal parameters p_a, p_p, and p_r stand for the value of field `p` and n_a, n_p, and n_r for the parameter to method `add`.

Finally, we give translation of a whole FJ-- program to a recursion scheme as in (the lower half of) Figure 6. Terminals include special symbols `br`, `fail` and `e` to denote branching caused by non-deterministic choice, failure and termination of program execution, respectively. Actions are terminals of arity 1. Rules are obtained by collecting all class translations (including *Object*, which provides only failing methods) and translation of the main statement. Non-terminal *End* stands for an initial continuation, which `e`nds the execution immediately. The sorts of non-terminals are easily computed by function *OType*, thanks to the fact that the bodies of most non-terminals are of the form

$$\lambda\langle \boldsymbol{x_1}\rangle.\cdots\lambda\langle \boldsymbol{x_n}\rangle.\lambda k.t$$

for some n and an applicative term t.

Properties of the Translation. We state properties of the translation. First, it is easy to see that the translation preserves typing, that is, a well typed FJ-- program translates to a well-formed recursion scheme.

Theorem 7. *If* $\vdash (CT, s) : C$*, then* $[\![(CT, s)]\!]$ *is a well-formed µHORS.*

Note that even when a given FJ-- program is not well typed, its translation may be a well-formed recursion scheme.

The resulting μHORS generates a tree that expresses as a path a sequence $a_1, a_2, \ldots$ of actions that can occur during the FJ-- program execution. More formally, the translation preserves the semantics of the program in the following sense:

Theorem 8. *Under a class table CT, $s \xrightarrow{a_1 \cdots a_n} r$ if and only if the tree generated by $[\![(CT, s)]\!]$ contains a path $a_1 \cdots a_n$ from the root (ignoring* $\mathtt{br}$*).*

It is easy to see as a corollary that (1) a program fails with *fail;* or `NoSuchMethodError` if and only if the tree generated by the translated μHORS contains a path that ends with `fail` and (2) a program terminates successfully if and only if the tree generated by the translated μHORS contains a path that ends with `e`.

We briefly discuss how properties listed at the beginning of the section can be verified. To verify reachability, we insert a special action to program points of interest and verify whether the generated tree contains the special action. We can verify a method invocation order by inserting distinct actions to the head of each method body. Although this method would be used to verify only a *global* method invocation history, it would be possible to modify the translation to verify an object-wise method invocation history, similarly to resource usage analysis [14] via HO model checking—see Kobayashi [22] for details.

Downcasts Given a whole FJ-- program with casts, we can transform it into an FJ-- program without casts, by adding a method *castToC* to class C, for every occurrence of $(C)e$. The definition of *castToC* for a class C is:

$$C\ \mathit{castToC}(\,)\{\mathtt{return\ this};\}$$

Note that *castToC* is defined only in C and its subclasses. So, downcasts will be transformed into an ill typed method invocation, although the recursion scheme after translation will be well formed thanks to the generation of failing methods. (It is easy to modify the translation so that failures due to `NoSuchMethodError` and `ClassCastException` are distinguished.)

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:

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

A program $P = M_1 \mid\mid M_2$ 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 (see examples given below). 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 of the language is given in Appendix B. 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. We can model programs accessing finite shared memory. For a one-bit shared variable x, prepare actions $\mathtt{read}_{x,i}$ and $\mathtt{write}_{x,i}$ for $i \in \{0,1\}$. Then, **let** $y = !x$ **in** M and $x := i; M$ can be expressed by $(\mathtt{read}_{x,0}; [0/y]M) \Box (\mathtt{read}_{x,1}; [1/y]M)$ and $\mathtt{write}_{x,i}; M$ respectively. Then, to check whether every (valid) execution of P satisfies a property ψ, it suffices to check that ψ is true of any path that represents valid read/write sequences, i.e. the most recent write action before each read action $\mathtt{read}_{x,i}$ must be $\mathtt{write}_{x,i}$, not $\mathtt{write}_{x,1-i}$.

Example 13. Let M be the following thread:

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

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 [22], 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:[5]

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

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 (in fact, with η-conversion, the transformation is exactly the same as the ordinary CPS transformation). The global action a is

[5] Here, for the sake of simplicity, we represent μHORS as an ordinary functional program with tree constructors. By applying λ-lifting, we get a system of top-level function definitions that conforms to the syntax of μHORS in Section 2.2.

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.

The transformation above preserves typing, in the sense that if P is a simply-typed program (where the global action a and non-determinism $\Box$ are given types $(\,)$ and $\forall\alpha.(\alpha \to \alpha \to \alpha)$ respectively), then $P^\dagger$ is a (well-typed) μHORS. To show this, let us define the translation of (simple) types by:

$$\begin{aligned}&\mathbf{unit}^\dagger = \mathtt{o}\\&(\tau_1 \to \tau_2)^\dagger = {\tau_1}^\dagger \to ({\tau_2}^\dagger \to \tau_{\mathtt{Ans}}) \to \tau_{\mathtt{Ans}},\end{aligned}$$

where $\tau_{\mathtt{Ans}}$ is the recursive type defined by:

$$\begin{aligned}\tau_{\mathtt{Ans}} &= \tau_{\mathtt{Gcont}} \to \mathtt{o}\\ \tau_{\mathtt{Gcont}} &= \tau_{\mathtt{Ans}} \to \mathtt{o}\end{aligned}$$

We extend the encoding to type environments by:

$$(x_1 : \tau_1, \dots, x_n : \tau_n)^\dagger = x_1 : {\tau_1}^\dagger, \dots, x_n : {\tau_n}^\dagger.$$

Then, we have:

Lemma 2. *Let M be a thread. If $\Gamma \vdash_{\mathtt{ST}} M : \tau$, then $\Gamma^\dagger \vdash M : (\tau^\dagger \to \tau_{\mathtt{Ans}}) \to \tau_{\mathtt{Ans}}$ holds.*

Proof. The proof follows by induction on the structure of M. If $M = a$, then we have $\Gamma^\dagger \vdash a : \mathtt{o} \to \mathtt{o}$ (recall that a is a unary terminal symbol after the translation). We also have $k : \mathbf{unit}^\dagger \to \tau_{\mathtt{Ans}}, g : \tau_{\mathtt{Gcont}} \vdash k\,\mathtt{e}\,g$ and $k : \mathbf{unit}^\dagger \to \tau_{\mathtt{Ans}}, g : \tau_{\mathtt{Gcont}} \vdash g\,(k\,\mathtt{e})$. Thus, we have $\Gamma^\dagger \vdash a^\dagger : (\mathbf{unit}^\dagger \to \tau_{\mathtt{Ans}}) \to \tau_{\mathtt{Ans}}$ as required. The other cases are straightforward; indeed, it is a standard result about the call-by-value CPS transformation [31].

Please note that in the above lemma, it is essential that we have recursive types; otherwise $a^\dagger$ cannot be typed. By assigning the type $\tau_{\mathtt{Ans}} \to \tau_{\mathtt{Ans}} \to \mathtt{o}$ to *Sched*, we have:

Corollary 9 *Let P be a program (of the form $M_1 \mid\mid M_2$). If $\emptyset \vdash_{\mathtt{ST}} P : (\,)$, then $\emptyset \vdash P^\dagger : \mathtt{o}$ holds.*

By the definition of the transformation above, it should be clear that 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. *Sched* is given type $\tau_{Thread} \to \tau_{Thread} \to \mathtt{o}$, where $\tau_{Thread} = \tau_{Gcont} \to \mathtt{o}$ and $\tau_{Gcont} = \tau_{Thread} \to \mathtt{o}$, where τ_{Thread} and τ_{Gcont} are the types of threads (which take a global continuation as an argument) global continuations respectively.

Example 14. Recall Example 13. The following is a hand-optimized version of μHORS obtained by translating the parallel composition of two copies of M.

```
S -> Sched M M.       Sched x y -> x (Sched y).
M g -> Sync Cr E g.  E g -> end.
Sync f k g -> lock (br (C1 f k g) (g (C1 f k))).
C1 f k g ->  f (C2 f k) g.
C2 f k g -> unlock(br (Sync f k g) (g (Sync f k))).
Cr k g -> enter (br (C3 k g) (g (C3 k))).
C3 k g -> exit (br (k g) (g k)).
```

As reported in Section 5, our model checker can verify that `enter` and `exit` can occur only alternately, as long as `lock` and `unlock` occur alternately, which implies that the two threads in the source program never enter critical sections simultaneously. □

Context-bounded model checking Qadeer and Rehof [37] 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 $Sched_\ell$ given below, which allows only ℓ context switches:

$$Sched_0\ x\ y \to \mathtt{e} \qquad Sched_{i+1}\ x\ y \to x\ (Sched_i\ y)$$

By the definition of the modified translation above, it should be trivial that P with context-bound ℓ has a sequence of global actions $a_1a_2\cdots a_n$ if and only if the tree generated by $P^{\dagger_\ell}$ has a path labeled with $a_1a_2\cdots a_n$ (with `br` ignored). Furthermore, the modified translation guarantees a stronger type-preservation property:

Lemma 3. *If* $\emptyset \vdash_{\mathtt{ST}} P : ()$*, then* $\emptyset \vdash P^{\dagger_\ell} : \mathtt{o}$ *is derivable (with intersection types but)* without using recursive types*.*

Thus, $P^{\dagger_\ell}$ is an ordinary HORS (extended with finite intersection types). As an immediate corollary of the above properties and the decidability of HORS model checking [33], we obtain that context-bounded model checking of multi-threaded higher-order programs is decidable.[6]

To show Lemma 3, we modify the encoding of types by:

$$\begin{aligned}&\mathbf{unit}^{\dagger_i} = \mathtt{o}\\ &(\tau_1 \to \tau_2)^{\dagger_i} = {\tau_1}^{\dagger_i} \to ({\tau_2}^{\dagger_i} \to \sigma_{\mathtt{Ans},i}) \to \sigma_{\mathtt{Ans},i}.\end{aligned}$$

where $\sigma_{\mathtt{Ans},i}$ is defined by:

$$\sigma_{\mathtt{Ans},0} = \top \qquad \sigma_{\mathtt{Ans},i+1} = \tau_{\mathtt{Gcont},i} \to \mathtt{o} \qquad \tau_{\mathtt{Gcont},i} = \sigma_{\mathtt{Ans},i} \to \mathtt{o}$$

The encoding is pointwise extended to that on type environments. The following lemma is analogous to Lemma 2.

[6] We have considered only programs with two threads, but this restriction can be easily relaxed by using the same technique as Qadeer and Rehof [37].

Lemma 4. *Let M be a thread and $\ell > 0$. If $\Gamma \vdash_{\mathtt{ST}} M : \tau$, then $\Gamma^{\dagger_\ell} \vdash M^\dagger : (\tau^{\dagger_\ell} \to \sigma_{\mathtt{Ans},\ell}) \to \sigma_{\mathtt{Ans},\ell}$ holds.*

Proof. The proof follows by induction on the structure of M. The only non-trivial case is when $M = a$. Let $\Gamma_1 = k : \mathbf{unit}^\dagger \to \sigma_{\mathtt{Ans},\ell}, g : \tau_{\mathtt{Gcont},\ell-1}$. Then, we obtain $\Gamma^{\dagger_\ell}, \Gamma_1 \vdash k\,\mathtt{e}\,g : \mathtt{o}$ immediately. As $\sigma_{\mathtt{Ans},1} \leq \sigma_{\mathtt{Ans},0} = \top$, we have $\sigma_{\mathtt{Ans},\ell} \leq \sigma_{\mathtt{Ans},\ell-1}$. Thus, we also obtain $\Gamma^{\dagger_\ell}, \Gamma_1 \vdash g(k\,\mathtt{e}) : \mathtt{o}$. Therefore, we have $\Gamma^{\dagger_\ell} \vdash a^\dagger : (\mathbf{unit}^\dagger \to \sigma_{\mathtt{Ans},\ell}) \to \sigma_{\mathtt{Ans},\ell}$ as required.

Lemma 3 follows as a corollary of the above lemma and the fact that $Sched_i$'s have the following (non-recursive) types:

$$\begin{array}{l} Sched_{2m} : \sigma_{\mathtt{Ans},m} \to \sigma_{\mathtt{Ans},m} \to \mathtt{o} \\ Sched_{2m+1} : \sigma_{\mathtt{Ans},m+1} \to \sigma_{\mathtt{Ans},m} \to \mathtt{o} \end{array}$$

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 reused some code of the model checker TRecS (for HORS without recursive types), but the core algorithm has been written from scratch (as the new algorithm is radically different from that of TRecS). 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 4, except for the following points. RTRecS first performs an equality-based flow analysis [35] 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 4) 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 [25].) 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 "answer" shows whether the property was judged to be satisfied (Y) or not (N). (The actual implementation returns recursive intersection types as a certificate in the former case, and a counter-example in the latter case.) 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. (For `ski1`, a counterexample was found before the SAT solver is called.) 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 given in Example 14, and `Thread2` is its variation, obtained by replacing the definition of cr in Example 13 by $cr\ x = (\,)\Box(cr\ x; \texttt{enter}; \texttt{exit})$. 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. `Ski1` and `Ski2` are implementations of SKI combinators in FJ [41], which were further translated to recursion schemes. `Ski1` and `Ski2` reduce SII and $SII(SII)$ respectively, and fails if the reduction terminates. As the reduction of SII terminates but $SII(SII)$ does not, the answers for `Ski1` and `Ski2` are N and Y respectively. The next six are list-manipulating programs (implemented as objects), which are small but non-trivial programs. (In fact, `L-filter` and `L-risers` are object-oriented versions of benchmark programs of the PMRS verification tool [34].) The program `L-append` repeatedly applies an `append` function (and never terminates), and fails if a method `hd` or `tl` is called for a null list object:

```
List loop_app(List l)
  { return this.loop_app(l.append(l)); }
```

The program `L-map` repeatedly applies a successor function to each element of a list (and never terminates). List constructors are generically defined so that `Object` is casted to `Nat` before each call of the successor function. The program fails if the successor function is applied to a non-number.

The program `L-app-map` repeatedly applies `map` and `append` functions:

```
class Main extends Object{ ...
 List loop_app_map(List l) {
  return this.loop_app_map
       (l.append(l).map(new Fun()));  }}
main() {
 return new Main().loop_app_map
      (new Cons(new S(new Z()),new Nil()));}
```

Here, `Fun` is an object representing a function that takes a natural number n and returns $n + n$, where natural numbers are also expressed as objects.

The program `L-even` creates a list of an even length in a non-deterministic manner[7], and fails if its length is odd.

The program `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). `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). `L-risers` creates a list of natural numbers in a non-deterministic manner, splits it into a

[7] We have extended FJ with non-deterministic branches.

programs	#lines	#rules	answer	#sat	time
$\mathcal{G}_1$	–	2	Y	27	0.001
$\mathcal{G}_2$	–	3	Y	49	0.002
`Thread`	–	9	Y	38,171	0.580
`Pred`	21	15	Y	157	0.005
`Ski1`	40	22	N	—	0.002
`Ski2`	40	25	Y	141	0.002
`L-append`	20	30	Y	165	0.006
`L-map`	43	182	Y	738	0.235
`L-app-map`	43	212	Y	1,546	0.391
`L-even`	25	87	Y	249	0.025
`L-filter`	59	122	Y	5,964	0.491
`L-risers`	73	64	Y	17,419	0.445
`Twofiles`	28	21	Y	739,867	13.86

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

list of lists consisting of non-decreasing sequences, and fails if a pattern match error occurs during the list processing.

`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 [22]. 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.

To see the effectiveness of the SAT-based approach to HO model checking, we have also compared RTRecS with two other model checkers for HORS: TRecS [21, 20] and GTRecS2 [19, 23, 26], which use radically different algorithms and cannot deal with μHORS. The results are summarized in Table 2.

The first two HORS `Twofiles-f`, `Fileocamlc` and `Order5` have been taken from [20], obtained by encoding resource usage verification problems [22]. `Mc91` and `Repeat` have been taken from [29], obtained by encoding reachability problems for functional programs using predicate abstraction. The other programs `Xhtmlf-id`, `Exp4-5`, and `Fibstring` have been taken from [30], [23], and [26], respectively. RTRecS is generally slower than TRecS for the problems obtained from program verification problems. RTRecS, however, outperforms TRecS for `Exp4-5` and `Fibstring`, which generate huge trees, and are hard cases for the TRecS algorithm. GTRecS2 has been tuned to deal with such cases, but RTRecS is actually faster for `Exp4-5`. Overall, the three model checkers are complementary to each other, which is encouraging given that the implementation of RTRecS is premature compared with the other two.

HORS	TRecS	GTRecS2	RTRecS
`Twofiles-f`	0.005	0.168	13.092
`Fileocamlc`	0.01	0.109	1.028
`Order5`	0.005	–	0.719
`Mc91`	0.07	0.379	–
`Repeat`	0.012	0.211	0.024
`Xhmtlf-id`	0.663	53.675	–
`Exp4-5`	–	2.422	0.367
`Fibstring`	–	0.210	0.284

Table 2. Comparison of Verification Times (in seconds). "–" indicates a time-out.

6 Discussion

We discuss related work and limitations of our approach.

6.1 Related Work

The model checking of HORS has recently emerged as a new technique for verification of higher-order programs [33, 22, 30, 34, 29]. Except Tsukada and Kobayashi's work [46], however, all the previous studies dealt with *simply-typed* recursion schemes, which are not suitable for modeling objects. Tsukada and Kobayashi [46] 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 sound (but necessarily incomplete) realistic procedure for model checking.

Several methods for model-checking functional programs have been proposed recently [40, 22, 30, 34, 44, 47], proposed recently [40, 22, 30, 34, 47], and some of them [40, 30, 34] 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 [40, 47] and are less precise.

There are previous studies on model checking of object-oriented programs [9, 12]. 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 (e.g., by identifying states having the same hash values [12]; see also `http://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/classification`, section "State Matching"). 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 is some similarity between the "state matching" used in Java PathFinder (see the URL above) and the use of equivalence relation $\sim$ in our model checking algorithm. Both of them merge states that are "equal". The fundamental difference is that Java PathFinder stops the reduction if an "equal" state is already visited, while our algorithm does not; rather, our algorithm generate even new states (e.g., $t_1t'_2$ and t'_1t_2 if t_1t_2 and $t'_1t'_2$ are visited and $t_1 \sim t'_1$) and explore reductions from them. This difference makes Java PathFinder efficient as a bug finding tool but not sound as a verification tool, and our tool slow but sound as a verification tool.

Another major approach to verification of object-oriented programs is to generate verification conditions from (annotated) programs, and then discharge them by theorem provers [2, 3]. Although there are some studies to infer invariants, this approach basically requires a lot of human intervention, such as annotations of pre/post-conditions for each method.

There are a number of studies on static analyses of object-oriented programs: see surveys [36, 8, 13, 39]. There are also studies of type systems [4, 5] and separation logic [36, 8] for verification of detailed properties about objects, which usually requires some annotations (e.g., typestate/aliasing annotations and/or pre/post-conditions of methods). Skalka [43] also proposed a method for automatically verifying temporal properties. His method is a combination of types and model checking, where a model is extracted through type inference and then it is model-checked. The first step (for extracting models through type inference) loses much information about source programs, while the first step in our method (the transformation of source programs into μHORS) loses no information. In general, the distinguished features of our method are: (i) it is fully automated yet very precise, (ii) it can generate a counter-example when a property is not satisfied, and (iii) it can verify temporal properties. For the last point, typestates [4, 5] can also verify some temporal properties, but they require special annotations. On the downside, our method is currently applicable only to functional objects. Rowe and Bakel [41] 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 [37, 17, 11], 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), and if so, whether the decidability can be obtained via reduction to HORS model checking as in Section 4.2, and whether tight complexity bounds can be obtained in that way. Note that the encoding in Section 4.2 does not give a good complexity bound (due to the use of CPS).

The implementation of our model checker uses the result of flow analysis to choose the equivalence relation $\sim$ and saturate the set $\mathcal{U}$. Our current implementation uses an imprecise, equality-based analysis, but in principle, it can be replaced by the result of any sound flow analysis, such as k-CFA [42].

6.2 Limitations

The fundamental limitations of our approach to automated program verification are:

1. Incompleteness of the recursive intersection type system with respect to the μHORS model checking problem.
   This means that there are correct programs that are not typable in the type system, in which case, our model checker never terminates. We have not yet obtained a clear characterization of such programs, but we expect that the class of such programs include those whose correctness proofs require numerical properties on multiple arguments (such as $x + y \geq z$).
2. Undecidability (only semi-decidability) of the typability in the recursive intersection type system (c.f. Theorem 3, 2).
   This means that even if a program is typable in the type system, there is no theoretical upper bound (even non-elementary one) on the time for model-checking to terminate.

To address the first limitation, we plan to treat numerical values as primitives and combine our method with predicate abstraction (as we did for verification of functional programs [29]). Extending the expressive power of recursive intersection types (so that we can use non-regular types) is also another interesting approach to address the first limitation. As for the second limitation, although we cannot bound the time complexity with respect to the program size, we can do so with respect to the size of the smallest certificate (i.e., the smallest witness type environment Γ such that $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$). Thus, if typical programs (and properties) have small certificates, then our approach may work well in practice (and the result of our preliminary experiments seem to indicate that this may indeed be the case). We should however refine the algorithm and implementation techniques further.

Our current verification tool for object-oriented programs only support FJ. The main obstacles to applying our approach to Java would be to support imperative field updates and concurrency; we believe that the other features (such as exceptions) can be easily handled either by encoding into FJ or by the translation to μHORS (c.f. encoding of control primitives of functional programs into

HORS [22, 20]). Since μHORS has recursive types, in theory, field updates and concurrency can also be encoded; we can use state passing for the former, and explicit representation of a scheduler for the latter, as discussed in Section 4.2. We expect, however, that more clever encoding and abstraction techniques are needed to make the whole verification work effectively in practice.

7 Conclusion

We have proposed a model checking procedure for μHORS, an extension of higher-order recursion schemes with recursive types, and shown its applications to verification of object-oriented programs and concurrent programs. The model-checking procedure is incomplete (due to the inherent undecidability of the model checking problem), but is relatively complete with respect to a type system with recursive intersection types. We have also implemented a prototype model checker and carried out preliminary experiments. Although there is a lot of work to be done to apply the proposed method to practice, we believe this is a good first step to get a fully-automated (i.e., requiring no annotations), sound (i.e., usable for verification, not just for bug finding) and precise software model checker for high-level programming languages that support objects, concurrency, and higher-order functions.

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

References

1. K. Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. *Logical Methods in Computer Science*, 3(3), 2007.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer-Verlag, 2005.
3. 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 *Lecture Notes in Computer Science*, pages 144–152. Springer-Verlag, 2005.
4. N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*, pages 227–244, 2008.
5. K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming (ECOOP'09)*, volume 5653 of *Lecture Notes in Computer Science*, pages 195–219, 2009.
6. K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Proceedings of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 415–438. Springer-Verlag, 1997.

7. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.
8. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 87–100, 2008.
9. 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.
10. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of computing with continuations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247, June 1993.
11. P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. *Formal Methods in System Design*, 40(2):206–231, 2012.
12. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000.
13. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of PASTE*, pages 54–61, 2001.
14. A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.
15. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
16. T. Johnsson. Lambda lifting: Treansforming programs to recursive equations. In *Proceedings of FPCA 85*, pages 190–203, 1985.
17. V. Kahlon. Reasoning about threads with bounded lock chains. In *Proceedings of CONCUR 2011*, volume 6901 of *Lecture Notes in Computer Science*, pages 450–465. Springer-Verlag, 2011.
18. 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.
19. N. Kobayashi. GTRecS2: A model checker for recursion schemes based on games and types. `http://www.kb.is.s.u-tokyo.ac.jp/~koba/gtrecs2/`, 2009.
20. N. Kobayashi. Model-checking higher-order functions. In *Proceedings of PPDP 2009*, pages 25–36. ACM Press, 2009.
21. N. Kobayashi. TRecS: A type-based model checker for recursion schemes. `http://www.kb.is.s.u-tokyo.ac.jp/~koba/trecs/`, 2009.
22. N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 416–428, 2009.
23. N. Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *Proceedings of FoSSaCS 2011*, volume 6604 of *Lecture Notes in Computer Science*, pages 260–274. Springer-Verlag, 2011.
24. N. Kobayashi. Model checking higher-order programs. *Journal of the ACM*, 60(3), 2013.
25. 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.
26. N. Kobayashi, K. Matsuda, and A. Shinohara. Functional programs as compressed data. In *Proceedings of PEPM 2012*, pages 121–130. ACM Press, 2012.

27. 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.
28. 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.
29. 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)*, 2011.
30. 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)*, pages 495–508, 2010.
31. A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi (summary). In *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer-Verlag, 1985.
32. M. L. Minsky. *Computation: Finite and infinite Machines*. Prentice-Hall, 1967.
33. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS 2006*, pages 81–90, 2006.
34. 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)*, pages 587–598, 2011.
35. J. Palsberg. Equality-based flow analysis versus recursive types. *ACM Transactions on Programming Languages and Systems*, 20(6):1251–1264, 1998.
36. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 75–86, 2008.
37. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proceedings of TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer-Verlag, 2005.
38. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.
39. V. Raman. Pointer analysis – a survey. Technical Report CS203, UC Santa Cruz, 2004.
40. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI 2008*, pages 159–169, 2008.
41. 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 *Lecture Notes in Computer Science*, pages 229–244, 2011.
42. O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.
43. C. Skalka. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation*, 21(3):239–282, 2008.
44. T. Terauchi. Dependent types from counterexamples. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 119–130, 2010.
45. Y. Tobita, T. Tsukada, and N. Kobayashi. Exact flow analysis by higher-order model checking. In *Proceedings of FLOPS 2012*, volume 7294 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2012.

46. T. Tsukada and N. Kobayashi. Untyped recursion schemes and infinite intersection types. In *Proceedings of FOSSACS 2010*, volume 6014 of *Lecture Notes in Computer Science*, pages 343–357. Springer-Verlag, 2010.
47. 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 *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag, 2010.

Appendix

A Proofs

A.1 Proof of Theorem 4

Below we assume that $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ and $\sim \subseteq \sim_\Gamma$. We also assume that $(\mathcal{X}_0, \mathcal{U}_0) \longrightarrow_\sim (\mathcal{X}_1, \mathcal{U}_1) \longrightarrow_\sim (\mathcal{X}_2, \mathcal{U}_2) \longrightarrow_\sim \cdots$ is a fair reduction sequence where $\mathcal{X}_0 = \{S\}$ and $\mathcal{U}_0 = \{(S, q_{\mathcal{B},0})\}$, and that $\mathcal{X}$ and $\mathcal{U}$ are $\bigcup_{i\in\omega} \mathcal{X}_i$ and $\bigcup_{i\in\omega} \mathcal{U}_i$ respectively. We let Θ_t be $\{\alpha_{[t],[t'_1],\ldots,[t'_\ell],q'} \mid (t\, t'_1 \cdots t'_\ell, q') \in \mathcal{U}\}$ in Section 3.1.

Lemma 5. *$\Gamma \vdash_{\mathcal{B}} t : q$ holds for every $(t, q) \in \mathcal{U}$.*

Proof. This follows from the properties that $\Gamma \vdash_{\mathcal{B}} S : q_{\mathcal{B},0}$ and that $\forall (t, q) \in \mathcal{U}_i.\Gamma \vdash_{\mathcal{B}} t : q$ implies $\forall (t', q') \in \mathcal{U}_{i+1}.\Gamma \vdash_{\mathcal{B}} t' : q'$. The latter can be shown by easy case analysis on the rule used for deriving $(\mathcal{X}_i, \mathcal{U}_i) \longrightarrow_\sim (\mathcal{X}_{i+1}, \mathcal{U}_{i+1})$.

- Case for (R-Const): It suffices to show that $\Gamma \vdash_{\mathcal{B}} a\, t_1 \cdots t_\ell : q$ and $\delta(q, a) = q_1 \cdots q_\ell$ imply $\Gamma \vdash_{\mathcal{B}} t_i : q_i$ for every i, which follows immediately from the typing rules.
- Case for (R-Fail): Trivial, as the reduction does not introduce any new pair of the form (t, q). (Actually, (R-Fail) is not applicable.)
- Case for (R-NT): This follows from the standard subject reduction property.
- Case for (R-Eq): It suffices to show that $\Gamma \vdash_{\mathcal{B}} t\, t_1 \cdots t_\ell : q$ and $t \sim t'$ imply $\Gamma \vdash_{\mathcal{B}} t'\, t_1 \cdots t_\ell : q$. By the assumption $\sim \subseteq \sim_\Gamma$, we have $\Gamma \vdash_{\mathcal{B}} t : \tau$ if and only if $\Gamma \vdash_{\mathcal{B}} t' : \tau$ for every τ. Thus, a derivation for $\Gamma \vdash_{\mathcal{B}} t'\, t_1 \cdots t_\ell : q$ can be constructed from that of $\Gamma \vdash_{\mathcal{B}} t\, t_1 \cdots t_\ell : q$ by replacing the derivation of $\Gamma \vdash_{\mathcal{B}} t : \tau$ with that of $\Gamma \vdash_{\mathcal{B}} t' : \tau$.

□

We say that a type derivation for $\Gamma \vdash_{\mathcal{B}} t : \tau$ is *normal* if it is derivable by using only judgments of the form $\Gamma' \vdash_{\mathcal{B}} u : (E, \alpha)$ where $\alpha \in \Theta_u$.

Lemma 6. *If $(t\, t_1 \cdots t_\ell, q) \in \mathcal{U}$, then there exists a normal derivation for $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} t : \bigwedge \Theta_{t_1} \to \cdots \to \bigwedge \Theta_{t_\ell} \to q$.*

Proof. By induction on the structure of t.

- Case $t = a$: By Lemma 5, $\Gamma \vdash_{\mathcal{B}} a\, t_1 \cdots t_\ell : q$ holds, so that we have $\delta(q, a) = q_1 \cdots q_\ell$ with $(t_i, q_i) \in \mathcal{U}$ for some $q_1, \ldots, q_\ell$. Thus, $q_i \in \Theta_{t_i}$ for each i. By the typing rule for constants, we have $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} a : \bigwedge \Theta_{t_1} \to \cdots \to \bigwedge \Theta_{t_\ell} \to q$ as required. Furthemore, $\bigwedge \Theta_{t_1} \to \cdots \to \bigwedge \Theta_{t_\ell} \to q \in \Theta_{[a]}$.
- Case $t = F$: This follows immediately from the definition of $\Gamma_{\mathcal{X},\mathcal{U},\sim}$.
- Case $t = s\, u$: By induction hypothesis, we have a normal derivation for:

$$\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} s : \bigwedge \Theta_u \to \bigwedge \Theta_{t_1} \to \cdots \to \bigwedge \Theta_{t_\ell} \to q.$$

For each $\sigma_1 \to \cdots \to \sigma_k \to q' \in \Theta_u$, there must exist $u_0, u_1, \ldots, u_k$ such that:

$$(u_0 u_1 \cdots u_k, q') \in \mathcal{U} \qquad u \sim u_0$$
$$\sigma_i = \bigwedge \Theta_{u_i} (1 \leq i \leq k)$$

Thus, we have $(uu_1 \cdots u_k, q') \in \mathcal{U}$. By induction hypothesis, we have a normal derivation for $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} u : \sigma_1 \to \cdots \to \sigma_k \to q'$. Therefore, we have a normal derivation for $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} su : \bigwedge \Theta_{t_1} \to \cdots \to \bigwedge \Theta_{t_\ell} \to q$ as required.

□

Proof of Theorem 4. As $S : q_{\mathcal{B},0} \in \Gamma_{\mathcal{X},\mathcal{U},\sim}$, it suffices to show $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma_{\mathcal{X},\mathcal{U},\sim}$. Suppose $F : (E, \alpha) \in \Gamma_{\mathcal{X},\mathcal{U},\sim}$, i.e., $\alpha \in \Theta_F$, and $\mathcal{R}(F) = \lambda x_1, \ldots, x_\ell.u$.

Then $E(\alpha)$ is of the form $\sigma_1 \to \cdots \to \sigma_\ell \to q$. By the definition of Θ_F, there must exist $u_0, u_1, \ldots, u_\ell$ such that:

$$F \sim u_0 \qquad (u_0 u_1 \cdots u_\ell, q) \in \mathcal{U} \qquad \sigma_i = \sigma_{[u_i]}$$

Thus, by the fairness of the reduction sequence, we have: $(Fu_1 \cdots u_m, q) \in \mathcal{U}$, which also implies:

$$([u_1/x_1, \ldots, u_\ell/x_\ell]t, q).$$

By Lemma 6, we have a normal derivation for

$$\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} [u_1/x_1, \ldots, u_\ell/x_\ell]t : q.$$

From the derivation of it, we obtain:

$$\Gamma_{\mathcal{X},\mathcal{U},\sim}, x : \sigma_{[u_1]}, \ldots, x : \sigma_{[u_\ell]} \vdash_{\mathcal{B}} t : q,$$

by replacing each node of the form $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} u_i : \tau_i$ with $\Gamma_{\mathcal{X},\mathcal{U},\sim}, x : \sigma_{[u_1]}, \ldots, x : \sigma_{[u_\ell]} \vdash_{\mathcal{B}} x_i : \tau_i$. Thus, we have $\Gamma_{\mathcal{X},\mathcal{U},\sim} \vdash_{\mathcal{B}} \lambda x_1, \ldots, x_\ell.t : \tau$ as required. □

A.2 Proof of Theorem 5

We first prove the following lemma.

Lemma 7. *Let $(\mathcal{X}_0, \mathcal{U}_0)$ be $(\{S\}, \{(S, q_{\mathcal{B},0})\})$. Suppose that $\sim'$ and $\sim$ be equivalence relations on* **Tm** *such that $\sim' \subseteq \sim$, and that*

$$(\mathcal{X}_0, \mathcal{U}_0) \longrightarrow_\sim (\mathcal{X}_1, \mathcal{U}_1) \longrightarrow_\sim (\mathcal{X}_2, \mathcal{U}_2) \longrightarrow_\sim \cdots, \text{ and}$$

$$(\mathcal{X}'_0, \mathcal{U}'_0) \longrightarrow_{\sim'} (\mathcal{X}'_1, \mathcal{U}'_1) \longrightarrow_{\sim'} (\mathcal{X}'_2, \mathcal{U}'_2) \longrightarrow_{\sim'} \cdots$$

are fair reduction sequences, with $(\mathcal{X}'_0, \mathcal{U}'_0) = (\mathcal{X}_0, \mathcal{U}_0)$. Then, $\Gamma_{\bigcup_i \mathcal{X}'_i, \bigcup_i \mathcal{U}'_i, \sim'} \sqsubseteq \Gamma_{\bigcup_i \mathcal{X}_i, \bigcup_i \mathcal{U}_i, \sim}$.

Proof. By the condition $\sim' \subseteq \sim$ and the fairness of the reductions, we have $(\bigcup_i \mathcal{X}'_i) \subseteq (\bigcup_i \mathcal{X}_i)$ and $(\bigcup_i \mathcal{U}'_i) \subseteq (\bigcup_i \mathcal{U}_i)$. Thus, the required condition follows from the definition of $\Gamma_{\mathcal{X},\mathcal{U},\sim}$. □

Theorem 5 follows as an immediate corollary of Theorem 4 and Lemma 7.

Proof of Theorem 5. Let $\sim' = (\sim \cap \sim_\Gamma)$, 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 $(\mathcal{X}'_0,\mathcal{U}'_0) = (\mathcal{X}_0,\mathcal{U}_0)$. By Theorem 4, we have $\Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S_{\mathcal{G}}) : q_{\mathcal{B},0}$ for $\Gamma' = \Gamma_{\bigcup_i \mathcal{X}'_i, \bigcup_i \mathcal{U}'_i, \sim'}$. By Lemma 7, we have $\Gamma' \sqsubseteq \Gamma_{\mathcal{X},\mathcal{U},\sim}$ as required. $\square$

B Operational Semantics of Multi-Threaded Programs

We define the transition relation $P \xrightarrow{\ell} P'$, where ℓ is either a global action or ϵ (which represents an internal computation). It means that program P is reduced to P' in one step, with an action ℓ. The set of values, ranged over by v, is given by:

$$v ::= (\,) \mid \texttt{fun}(f, x, M)$$

The relation $P \xrightarrow{\ell} P'$ is inductively defined by:

$$\frac{}{(\texttt{fun}(f,x,M))V \xrightarrow{\epsilon} [\texttt{fun}(f,x,M)/f, V/x]M}$$

$$\frac{i \in \{1,2\}}{M_1 \Box M_2 \xrightarrow{\epsilon} M_i}$$

$$\frac{M \xrightarrow{\ell} M'}{MN \xrightarrow{\ell} M'N}$$

$$\frac{N \xrightarrow{\ell} N'}{MN \xrightarrow{\ell} MN'}$$

$$\frac{M \xrightarrow{\ell} M'}{M \,||\, N \xrightarrow{\ell} M' \,||\, N}$$

$$\frac{N \xrightarrow{\ell} N'}{M \,||\, N \xrightarrow{\ell} M \,||\, N'}$$

We write $M \xrightarrow{a_1 \cdots a_n} N$ if:

$$M (\xrightarrow{\epsilon})^* \xrightarrow{a_1} (\xrightarrow{\epsilon})^* \cdots (\xrightarrow{\epsilon})^* \xrightarrow{a_n} (\xrightarrow{\epsilon})^* N.$$