Types and Higher-Order Recursion Schemes for Verification of Higher-Order Programs

Naoki Kobayashi
Tohoku University
koba@ecei.tohoku.ac.jp

Abstract

We propose a new verification method for temporal properties of higher-order functional programs, which takes advantage of Ong's recent result on the decidability of the model-checking problem for higher-order recursion schemes (HORS's). A program is transformed to an HORS that generates a tree representing all the possible event sequences of the program, and then the HORS is model-checked. Unlike most of the previous methods for verification of higher-order programs, our verification method is sound and complete. Moreover, this new verification framework allows a smooth integration of abstract model checking techniques into verification of higher-order programs. We also present a type-based verification algorithm for HORS's. The algorithm can deal with only a fragment of the properties expressed by modal μ-calculus, but the algorithm and its correctness proof are (arguably) much simpler than those of Ong's game-semantics-based algorithm. Moreover, while the HORS model checking problem is n-EXPTIME in general, our algorithm is linear in the size of HORS, under the assumption that the sizes of types and specifications are bounded by a constant.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

1. Introduction

With the increasing importance of software reliability, program verification techniques have been studied extensively. There are still limitations in the current verification technology, however. Software model checking [3–5] has been mainly applied to imperative programming languages, and applications to programming languages with more dynamic control, such as higher-order languages with dynamic allocation of resources (such as heap memory), have been limited. For higher-order programs, *type systems* have been recognized as effective techniques for program verification. However, they either require explicit type annotations (as in dependent type systems), or suffer from many false alarms.

In this paper, we propose a novel verification technique for temporal properties of higher-order programs. We consider the problem of resource usage verification [19] for higher-order functional languages with dynamic resource creation and access primitives. The goal of the verification is to check that each dynamically created resource is accessed in a proper manner (like "an opened file is eventually closed, and it is not read or written after being closed"). Assertion-based model-checking problems (like "$X > 0$ holds at program point p") can also be recasted as the resource verification problem, by regarding an assertion failure as an access to a global resource. (For example, "`assert(b)`" can be transformed into "`if b then skip else fail`," where `fail` is an action to the global resource. Then the problem of checking lack of assertion failures is reduced to the resource usage verification problem of checking whether the `fail` action occurs.)

Our verification technique is built on the recent result on model checking of *higher-order recursion schemes* (HORS's, for short) [29]. A higher-oder recursion scheme is a grammar for describing an infinite tree. HORS is a generalization of regular tree grammars; they are described by HORS's of order 0. Ong [29] has recently shown the decidability of the problem of checking whether the infinite tree generated by $\mathcal{G}$ satisfies ψ, given a modal μ-calculus formula ψ and an HORS $\mathcal{G}$.

The first main idea of this paper is to transform a higher-order functional program into an HORS that produces an infinite tree, each of whose path (from the root) corresponds to a possible access sequence to each resource. By the transformation, the problem of checking a regular property of resource-wise access behaviors of a functional program is reduced to that of checking the corresponding regular property of the infinite tree generated by the HORS. The latter can be solved by Ong's model checking algorithm for HORS [29]. For programs having only resources and functions as values, the transformation into HORS is achieved by CPS conversion and λ-lifting, along with an additional trick to extract "resource-wise" access behavior. For programs with ordinary values (such as integers), we can apply the technique of predicate abstractions and counter-example-guided abstraction refinement. The resulting verification framework is sketched in Figure 1. Given a source program, we first apply CPS conversion and λ-lifting to get a system of top-level function definitions (Step 1). We then apply predicate abstractions to get a higher-order boolean program (Step 2). It is then converted to HORS (Steps 3 and 4), and the HORS is model-checked (Step 5). If the model checking fails, a counterexample is investigated, and the abstraction is refined (Steps 6 and 7).

The second main idea of this paper is to use *types* for model-checking HORS, instead of Ong's algorithm based on game semantics. For a fragment of modal μ-calculus (for describing safety properties, which are sufficient for the purpose of resource usage verification), we develop an intersection type system that is sound and *complete*, in the sense that an HORS is well-typed if and only if the HORS satisfies the given property. Thus, a type inference algo-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-379-2/09/01. . . $5.00

rithm for that type system serves as an alternative model checking algorithm for HORS. The resulting verification algorithm is (arguably) simpler and easier to understand than Ong's algorithm [29]. Moreover, although the resulting algorithm suffers from the same worst-case time complexity (i.e., n-EXPTIME for order-n HORS) as Ong's algorithm, our algorithm runs in time linear in the size of the rules of HORS, under the assumption that the size of types and specifications are bound by a constant. An additional advantage of the type-based verification of HORS is that it makes easier to compare the new resource usage verification method with previous type-based methods for resource usage verification (or, typestate checking) [12, 19, 32], and combine them together. In fact, previous type systems for resource usage verification may be viewed as a restriction of our intersection type system, with a limited form of intersection types. Thus, we can first apply previous type systems to verification of HORS, and then gradually refine types (by allowing more flexible intersection types). Since the full intersection type system is complete, the refinement process will eventually terminate and produce a yes/no answer.

The whole verification framework thus obtained is an integration of model checking and type-based verification techniques. We use model-checking techniques (predicate abstraction with counter-example-guided abstraction refinement) to abstract information about *data* (or base values), and type-based techniques to abstract information about *control* (or functions).

The rest of this paper is structured as follows. Section 2 reviews the definitions and the decidability result on higher-order recursion schemes. Section 3 defines the resource usage verification problem for a language having only resources and functions as values, and shows the reduction of the resource usage verification to the model-checking problem of HORS. Section 4 discusses how to extend the verification method to deal with source programs (having ordinary data such as integers), by using techniques of abstract model checking. Section 5 presents an intersection type system that is sound and complete for (safety properties of) HORS, and discusses its type inference algorithm. Section 6 (informally) compares the intersection type system with existing type systems for resource usage verification. Section 7 discusses related work, and Section 8 concludes.

Omitted proofs and more examples are found in the extended version of this paper [25].

2. Preliminaries

2.1 Higher-Order Recursion Schemes

This subsection reviews the definition of higher-order recursion schemes and decidability results [29].

A higher-order recursion scheme (HORS, for short) is a grammar for describing an infinite tree. The set of *types* is defined by:

$$\kappa ::= \mathtt{o} \mid \kappa_1 \to \kappa_2$$

Intuitively, $\mathtt{o}$ describes trees, while $\kappa_1 \to \kappa_2$ describes a function that takes an entity of type κ_1 and returns an entity of type κ_2. The *order* and *arity* of κ, written $order(\kappa)$ and $arity(\kappa)$ respectively, are defined by:

$$\begin{array}{l} order(\mathtt{o}) = 0 \\ order(\kappa_1 \to \kappa_2) = max(order(\kappa_1) + 1, order(\kappa_2)) \\ arity(\mathtt{o}) = 0 \\ arity(\kappa_1 \to \kappa_2) = arity(\kappa_2) + 1 \end{array}$$

A (deterministic) HORS $\mathcal{G}$ is a 4-tuple $(\Sigma, \mathcal{N}, \mathcal{R}, S)$, where

- Σ is a mapping from a finite set of symbols called *terminals* to types of order 0 or 1.
- $\mathcal{N}$ is a mapping from a finite set of symbols called *non-terminals* to types.

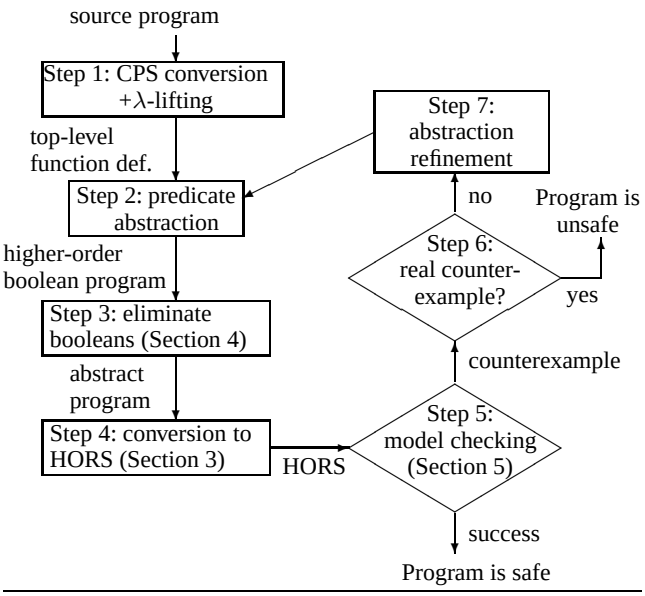

Figure 1. Verification Framework

- $\mathcal{R}$ is a set of rewriting rules of the form:

$$\{F_1\ \widetilde{x}_1 \to t_1, \ldots, F_n\ \widetilde{x}_n \to t_n\}$$

  Here, $\widetilde{x}$ abbreviates a sequence of variables, and t is a term constructed from non-terminals, terminals, and variables (see below). There must be exactly one rule of the form $F\ \widetilde{x} \to t$ for each non-terminal $F \in \mathcal{N}$.[1]
- S is a special non-terminal called the *start symbol*.

We require that $\mathcal{N}(S) = \mathtt{o}$. The set of (typed) terms is defined in the standard manner: A symbol (i.e., a terminal, non-terminal, or variable) of type κ is a term of type κ. If terms t_1 and t_2 have types $\kappa_1 \to \kappa_2$ and κ_1 respectively, then $t_1\ t_2$ is a term of type κ_2. For each rewriting rule $F\ \widetilde{x} \to t$, $F\ \widetilde{x}$ and t must be terms of type $\mathtt{o}$. The *order* of an HORS is the highest order of its non-terminals.

By abuse of notation, we often write $a \in \Sigma$ and $F \in \mathcal{N}$ for $a \in dom(\Sigma)$ and $F \in dom(\mathcal{N})$.

The rewriting relation $\longrightarrow_{\mathcal{G}}$ is defined inductively by:

- $F\ \widetilde{s} \longrightarrow_{\mathcal{G}} [\widetilde{s}/\widetilde{x}]t$ if $F\ \widetilde{x} \to t$ is a rule of $\mathcal{G}$.
- If $t \longrightarrow_{\mathcal{G}} t'$, then $ts \longrightarrow_{\mathcal{G}} t's$ and $st \longrightarrow_{\mathcal{G}} st'$.

We omit the subscript $\mathcal{G}$ if it is clear from the context.

We need auxiliary definitions to define the (possibly infinite) tree generated by $\mathcal{G}$. Let n be the maximum arity of symbols in Σ. A (possibly infinite) tree over Σ is a partial function t from $\{0, \ldots, n-1\}^*$ to $dom(\Sigma)$, such that: (1) $\epsilon \in dom(t)$; (2) the domain of t is closed under prefix operations; and (3) if $t(w) = a$ and $arity(\Sigma(a)) = m$, then $\{j \mid w\ j \in dom(t)\} = \{0, \ldots, m-1\}$. A (possibly infinite) sequence π over $\{0, \ldots, n-1\}$ is a *path* of t if every finite prefix of π is in $dom(t)$.

We often use the usual term representation for trees. For example, we write $\mathtt{br}\ \mathtt{a}\ \mathtt{b}$ for the tree:

$$\{\epsilon \mapsto \mathtt{br}, 0 \mapsto \mathtt{a}, 1 \mapsto \mathtt{b}\}.$$

[1] This restriction is not present for *non-deterministic* HORS. We consider only deterministic HORS's in this paper.

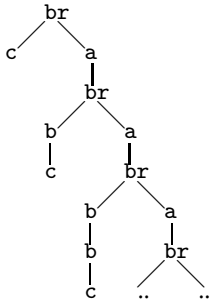

Figure 2. The value tree $[\![\mathcal{G}_0]\!]$

Given a term t, we define a (finite) tree $t^\perp$ by:

$$t^\perp = \begin{cases} f & \text{if } t \text{ is a terminal } f \\ {t_1}^\perp {t_2}^\perp & \text{if } t \text{ is of the form } t_1 t_2 \text{ and } {t_1}^\perp \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

For example, $(f\ (F\ a)\ b)^\perp = f\ \perp\ b$. Let $\sqsubseteq$ be the partial order on $dom(\Sigma) \cup \{\perp\}$ defined by $\forall a \in dom(\Sigma).\perp \sqsubseteq a$. It is extended to the partial order on trees by: $t \sqsubseteq s$ iff $\forall w \in dom(t).(w \in dom(s) \wedge t(w) \sqsubseteq s(w))$. For example, $\perp \sqsubseteq f\ \perp\ \perp \sqsubseteq f\ \perp\ b \sqsubseteq f\ a\ b$. For a set T of trees, we write $\bigsqcup T$ for the least upper bound of elements of T with respect to $\sqsubseteq$.

The *value tree* of $\mathcal{G}$, written $[\![\mathcal{G}]\!]$, is defined by:

$$[\![\mathcal{G}]\!] = \bigsqcup\{t^\perp \mid S \longrightarrow_{\mathcal{G}}^* t\}.$$

EXAMPLE 2.1. *Consider the recursion scheme* $\mathcal{G}_0 = (\Sigma, \mathcal{N}, \mathcal{R}, S)$*, where:*

$$\begin{array}{l} \Sigma = \{\mathtt{br} : \mathtt{o} \to \mathtt{o} \to \mathtt{o}, \mathtt{a}, \mathtt{b} : \mathtt{o} \to \mathtt{o}, \mathtt{c} : \mathtt{o}\} \\ \mathcal{N} = \{S : \mathtt{o}, F : \mathtt{o} \to \mathtt{o}\} \\ \mathcal{R} = \{S \to F\ \mathtt{c}, \quad F\ x \to \mathtt{br}\ x\ (\mathtt{a}(F(\mathtt{b}(x))))\} \end{array}$$

S can be reduced as follows.

$$\begin{array}{rcl} S & \longrightarrow & F\ \mathtt{c} \\ & \longrightarrow & \mathtt{br}\ \mathtt{c}\ (\mathtt{a}(F(\mathtt{b}(\mathtt{c})))) \\ & \longrightarrow & \mathtt{br}\ \mathtt{c}\ (\mathtt{a}(\mathtt{br}\ (\mathtt{b}(\mathtt{c}))\ (\mathtt{a}(F(\mathtt{b}(\mathtt{b}(\mathtt{c}))))))) \\ & \longrightarrow & \cdots \end{array}$$

The value tree $[\![\mathcal{G}_0]\!]$ is shown in Figure 2. □

In general, the value tree of HORS may have a non-regular structure. For example, in the example above, the set of paths from the root of the value tree is $\{\mathtt{br}(\mathtt{a}\ \mathtt{br})^n \mathtt{b}^n \mathtt{c} \mid n \geq 0\}$. According to the recent result [29], however, given an HORS $\mathcal{G}$ and a regular property ψ, it is decidable whether the value tree of $\mathcal{G}$ satisfies ψ.

THEOREM 2.1 (Ong [29]). *Let $\mathcal{G}$ be an HORS of order n, and ψ be a formula of modal μ-calculus. The problem of checking whether $[\![\mathcal{G}]\!]$ satisfies ψ is n-EXPTIME-complete.*

REMARK 2.1. *In this paper, we only consider HORS's whose value trees do not contain $\perp$. That condition is indeed satisfied by the higher-order recursion schemes generated by our verification method discussed in Section 3.*

2.2 Tree Automata for Infinite Trees

We use tree automata for describing properties of (the value tree of) HORS, instead of modal μ-calculus formulas. We recall below some basic definitions of (top-down) tree automata for infinite trees. See [33] for a good survey of logics and automata for infinite trees.

A Büchi automaton is a 5-tuple $M = (Q, \Sigma, q_S, \Delta, Q_F)$ where:

- Q is a finite set of states.
- Σ is a mapping from a finite set of input symbols to types of order 1.[2]
- $q_S (\in Q)$ is a state called an *initial state*.
- $\Delta \subseteq \bigcup_k (Q \times dom(\Sigma) \times Q^k)$, such that if $(q, a, q_1, \ldots, q_k) \in \Delta$, then $arity(\Sigma(a)) = k$ (that is, the arity of a is respected).
- $Q_F (\subseteq Q)$ is a set of states, called *accepting states*.

A run of M on the tree t over Σ is a tree ρ over Q, such that $\rho(\epsilon) = q_S$ and $(\rho(w), t(w), \rho(w0), \ldots, \rho(w(k-1))) \in \Delta$ for any $w \in dom(t)$ and $k = \Sigma(t(w))$. The run ρ is successful if for each infinite path π, there are infinitely many prefixes $w_0, w_1, \ldots$ of π such that $\rho(w_i) \in Q_F$. (In other words, a run is successful if every infinite path of the run visits accepting states infinitely often.) The automaton M accepts t if there is a successful run of M on t.

Actually, in this paper, we will consider only the case where $Q_F = Q$, so that M accepts t if there is a run of M on t. This is called a *trivial automaton* in [1]. Such automata can be used to express safety properties, meaning that a bad thing never happens (when a tree is traversed from the root).

3. Resource Usage Verification Based on Higher-Order Recursion Scheme

In this section, we define the problem of resource usage verification and show how to reduce it to the model-checking problem for an HORS. The target language for resource usage verification in this section is a simply-typed functional language with resources and non-deterministic branches. Extensions of the language with values (such as booleans and integers) and conditional branches will be discussed in Section 4.

3.1 Overview

The idea of reducing a resource usage verification problem to an HORS model-checking problem is to transform a program into an HORS that generates a tree denoting all the possible resource access sequences. For example, consider the following program (written in OCaml-like language):

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

It opens a read-only file "foo," reads it several times, and then closes it. We shall transform the above program to a recursion scheme that generates a tree like the one in Figure 3. In the figure, the terminals `r` and `c` stand for read and close operations, and the terminal $\nu^{\mathtt{r}^*\mathtt{c}}$ stands for creation of a read-only file (descriptor). `br` and $\star$ denote a non-deterministic branch and a program termination respectively. The tree can be generated by the following higher-order recursion scheme:

$$\begin{array}{rcl} G\ x\ k & \to & \mathtt{br}\ (\mathtt{c}(k))\ (\mathtt{r}(G\ x\ k)) \\ S & \to & \nu^{\mathtt{r}^*\mathtt{c}}(G\ \mathtt{d}\ \star) \end{array}$$

Note that this is essentially a CPS-transformation! G has an additional parameter k for expressing the rest of the computation, and `close(x)` has been replaced by `c`, followed by k. Unlike in the usual CPS program, however, resource access primitives `read` and `close` have been replaced by the terminals (or, tree constructors) `r`

[2] Σ may also be regarded as a mapping from input symbols to non-zero arities. We require that the arity is non-zero, so that every branch of a tree has an infinite path.

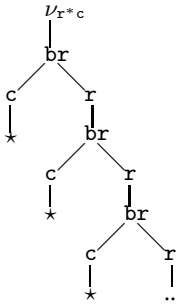

Figure 3. A tree that represents possible resource access sequences

and c. The conditional branch in g has been replaced by the terminal br, which expresses a non-deterministic branch. In the recursion scheme above, d is a dummy symbol; the first argument of G above is in fact unnecessary.

As the example above shows, for a program that accesses a single resource, an HORS generating the resource access tree is obtained by CPS transformation followed by λ-lifting (that moves local function definitions up to top-level; this is necessary since higher-order recursion schemes do not have local rewriting rules).

An additional trick is required, however, for a program that creates and accesses more than one (possibly an infinite number of) resources. For example, consider the following program:

```
let rec f x y = if b then close(x);close(y)
                else read(x);write(y);f x y in
let r1 = open_in "foo" in
let r2 = open_out "bar" in f r1 r2
```

Since we need to verify that each of r1 and r2 will be accessed in a valid manner, we transform the program into an HORS that generates a tree that represents *resource-wise* access sequences. The required HORS is:

$$\begin{array}{rcl} F\ x\ y\ k & \to & \mathtt{br}\ (x\ \mathtt{c}\ (y\ \mathtt{c}\ k))\ (x\ \mathtt{r}\ (y\ \mathtt{w}\ (F\ x\ y\ k))) \\ S & \to & \mathtt{br}\ (C_1\ K)\ (\nu^{\mathtt{r}^*\mathtt{c}}(C_1\ I)) \\ C_1\ x & \to & \mathtt{br}\ (F\ x\ K\ \star)\ (\nu^{\mathtt{w}^*\mathtt{c}}(F\ x\ I\ \star)) \\ I\ x\ k & \to & x\ k \\ K\ x\ k & \to & k \end{array}$$

Here, upon a resource creation, the resource is non-deterministically instantiated to I or K, of type $(\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$ (see the rules for S and C_1). When the resource is instantiated to I, it is recorded that a resource has been created, by ν. I and K take a resource access operation x as an argument; I attaches x to the access tree, while K just ignores x. Intuitively, I is a resource for which we should keep track of access sequences, while K is a resource for which we should ignore access sequences. An access to a resource x (which is bound to either I or K) is now transformed into $x\ a\ k$, where a is the name of the access primitive, and k is the tree representing the rest of the computation. If x is I, then it is evaluated to $a(k)$; otherwise (if x is K), it is evaluated to k, with a being ignored.

The tree generated by the above recursion scheme is shown in Figure 4. The four sub-trees marked by dashed boxes express possible access sequences obtained by keeping track of different resources. In t_1, both the resources are ignored. In t_2, the write-only file is focused on, while in t_3, the read-only file is focused on. Both files are focused on in t_4. For the purpose of resource usage verification, we need to check only t_2 and t_3; we can blindly accept the subtrees t_1 and t_4, which either contain no ν, or more than one ν.

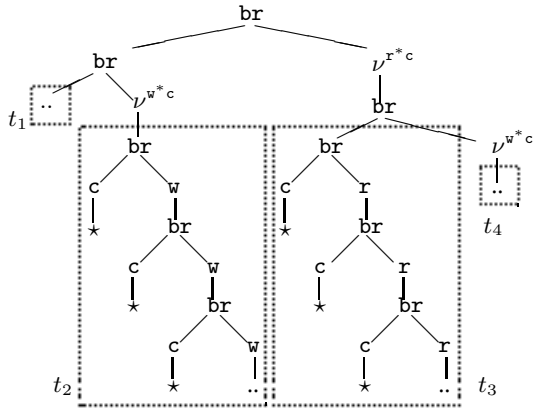

Figure 4. A tree that represents access sequences for two resources

In the rest of this section, we first introduce a functional language and formally define the resource usage verification problem in Section 3.2. We then define the transformation from programs into HORS's, and prove the correctness in Section 3.3.

3.2 Resource Usage Verification

We introduce a functional language and the resource usage verification problem for it. Since the CPS transformation and λ-lifting have been well studied, we consider programs that are already in the continuation-passing style, having only top-level function definitions.[3] The language we consider here has essentially the same expressive power as Igarashi and Kobayashi's λ-calculus with resources [19]. (Their calculus does have booleans, but their type-based analysis does not distinguish between **true** and **false**, treating a conditional as a non-deterministic choice.)

A *program* D is a set of *function definitions* $\{F_1\ \widetilde{x}_1 = e_1, \ldots, F_n\ \widetilde{x}_n = e_n\}$, where F_i denotes a defined function symbol, and e ranges over the set of expressions, defined by:

$$e ::= \star \mid x \mid F \mid e_1 e_2 \mid \mathbf{if}{*}\ e_1\ e_2 \mid \mathbf{new}^L\ e \mid \mathbf{acc}_a\ x\ e$$

We assume that $F_1, \ldots, F_n$ are different from each other, and that any program D contains exactly one definition for S (which is the "main" function), of the form $S = e$.

The expression $\star$ is the unit-value. The expression $e_1 e_2$ applies the function e_1 to e_2, and $\mathbf{if}{*}\ e_1\ e_2$ non-deterministically executes e_1 or e_2. The expression $\mathbf{new}^L\ e$ creates a resource that should be used according to the specification L, and passes it to e (which is a function that takes a resource as an argument). Here, L is the set of access sequences that may occur until the program terminates. For example, the specification for read-only files is $\{\mathtt{c}, \mathtt{r}\ \mathtt{c}, \mathtt{r}\ \mathtt{r}\ \mathtt{c}, \ldots\}$. Note that the set does not contain the sequence r, since files must be closed before the program termination. In this paper, we assume that L is a regular language.[4] $\mathbf{acc}_a\ x\ e$ applies an operation of name a to the resource x, and then evaluates e. a ranges over a finite set of the names of resource access primitives (like r, w, and c).

We write $[e_1/x]e_2$ for the expression obtained by replacing all the occurrences of x in e_2 with e_1. A sequence of expressions $e_1, \ldots, e_n$ is often abbreviated to $\widetilde{e}$.

[3] Note that the CPS transformation and λ-lifting do not change the resource access sequences that we are interested in.

[4] Most of the previous methods for resource usage or typestate verification [12, 32] impose essentially the same restriction. Replacing the class of regular languages with that of context-free languages make the problem undecidable.

We consider only "well-typed" programs below. The set of types is given by:

$$\tau \text{ (types)} ::= \mathbf{R} \mid \mathbf{unit} \mid \tau_1 \to \tau_2$$

Here, $\mathbf{R}$ is the type of resources and $\mathbf{unit}$ is the type of the unit value.

A type environment, denoted by Γ, is a mapping from variables (including function names $F_1, \ldots, F_n$) to types. The type judgment relation $\Gamma \vdash e : \tau$ for expressions is the least relation closed under the following rules:

$$\frac{}{\Gamma \vdash \star : \mathbf{unit}} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma, F : \tau \vdash F : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \mathbf{unit} \quad \Gamma \vdash e_2 : \mathbf{unit}}{\Gamma \vdash \mathbf{if}* \, e_1 \, e_2 : \mathbf{unit}}$$

$$\frac{\Gamma \vdash e : \mathbf{R} \to \mathbf{unit}}{\Gamma \vdash \mathbf{new}^L \, e : \mathbf{unit}} \qquad \frac{\Gamma \vdash e_1 : \mathbf{R} \quad \Gamma \vdash e_2 : \mathbf{unit}}{\Gamma \vdash \mathbf{acc}_a \, e_1 \, e_2 : \mathbf{unit}}$$

A program $D = \{F_1 \, \widetilde{x} = e_1, \ldots, F_n \, \widetilde{x} = e_n\}$ is well-typed, if there exists a type environment $\Gamma = F_1 : \widetilde{\tau}_1 \to \mathbf{unit}, \ldots, F_n : \widetilde{\tau}_n \to \mathbf{unit}$ such that $\Gamma, \widetilde{x} : \widetilde{\tau}_i \vdash e_i : \mathbf{unit}$ holds for each i; Here, $\widetilde{\tau} \to \mathbf{unit}$ and $\widetilde{x} : \widetilde{\tau}$ are abbreviations of $\tau_1 \to \cdots \to \tau_n \to \mathbf{unit}$ and $x_1 : \tau_1, \ldots, x_n : \tau_n$ respectively.

Note that all the defined function symbols have types of the form $\widetilde{\tau} \to \mathbf{unit}$. That condition is met by programs in CPS form.

REMARK 3.1. *The above language contains programs that are not images of CPS transformation. The transformation in the rest of this section applies not only to images of CPS transformation but to the whole language.*

We now define an operational semantics by using the relation $(H, e) \longrightarrow_D (H', e')$ on run-time states. A *run-time state* is expressed by a pair (H, e), where e is the current expression, and H maps each resource to a pair (w, L), where L is the specification of the resource, and w is the sequence of accesses that have occurred so far.

The relation $(H, e) \longrightarrow_D (H', e')$ is the least relation closed under the following rules.

$$(H, F \, \widetilde{e'}) \longrightarrow_{D \cup \{F \, \widetilde{x} = e\}} (H, [\widetilde{e'}/\widetilde{x}]e)$$

$$(H, \mathbf{if}* \, e_1 \, e_2) \longrightarrow_D (H, e_1)$$

$$(H, \mathbf{if}* \, e_1 \, e_2) \longrightarrow_D (H, e_2)$$

$$(H, \mathbf{new}^L \, e) \longrightarrow_D (H \cup \{x \mapsto (\epsilon, L)\}, e \, x) \qquad (x \notin dom(H))$$

$$(H \uplus \{x \mapsto (w, L)\}, \mathbf{acc}_a \, x \, e) \longrightarrow_D (H \cup \{x \mapsto (w \, a, L)\}, e)$$

Here, $H \uplus \{x \mapsto (w, L)\}$ is defined to be $H \cup \{x \mapsto (w, L)\}$ only if $x \notin dom(H)$. $w \, a$ is the concatenation of the two strings w and a.

REMARK 3.2. *Note that the reduction is allowed only at the top-level; For example, $(H, F(\mathbf{if}* \, e_1 \, e_2)) \longrightarrow_D (H, F \, e_1)$ is not allowed. This is not a limitation, since we are dealing with programs in CPS form.*

The following is the standard type soundness property.

LEMMA 3.1. *Suppose D is well-typed. If $(\emptyset, S) \longrightarrow_D^* (H, e)$, then either e is $\star$ or $(H, e) \longrightarrow_D (H', e')$ for some (H', e').*

We now define the problem of resource usage verification. The first condition says that no invalid resource access occurs, and the second condition says that all the required accesses must have occurred when the program terminates.

DEFINITION 3.1 (resource safety, resource usage verification). *A (well-typed) program D is resource-safe if the following conditions are satisfied.*

1. *If $(\emptyset, S) \longrightarrow_D^* (H, e)$ and $H(x) = (w, L)$, then $ww' \in L$ for some w'.*
2. *If $(\emptyset, S) \longrightarrow_D^* (H, \star)$ and $H(x) = (w, L)$, then $w \in L$.*

Resource usage verification is the problem of checking whether a given (well-typed) program is resource-safe or not.

3.3 Transformation into HORS

We now give a transformation of a program D into a pair consisting of an HORS $\mathcal{H}(D)$ and a tree automaton $\mathcal{M}(D)$ (for infinite trees), such that D is resource-safe if and only if $[\![\mathcal{H}(D)]\!]$ is accepted by $\mathcal{M}(D)$.

Let $\mathbf{Fnames}(D)$, $\mathbf{Anames}(D)$, and $\mathbf{Specs}(D)$ be the sets of defined function symbols, access primitive names (such as $\mathtt{r}$, $\mathtt{w}$), specifications (L that appears in the form $\mathbf{new}^L$) respectively. The transformation $\mathcal{H}$ from programs to HORS's is defined by:

$$\begin{array}{l}
\mathcal{H}(D) = (\Sigma, \mathcal{N}, \mathcal{R}, S) \\
\text{where:} \\
\Sigma = \mathbf{Anames}(D) \cup \{\nu^L \mid L \in \mathbf{Specs}(D)\} \cup \{\mathtt{call}, \mathtt{t}, \mathtt{br}\} \\
\mathcal{N} = \mathbf{Fnames}(D) \cup \{\mathbf{new}^L \mid L \in \mathbf{Specs}(D)\} \\
\qquad \cup \{\mathbf{acc}_a \mid a \in \mathbf{Anames}(D)\} \cup \{\mathbf{if}*, I, K, \star\} \\
\mathcal{R} = \{F \, \widetilde{x} \to \mathtt{call}(e) \mid F \, \widetilde{x} = e \in D\} \\
\qquad \cup \{\mathbf{acc}_a \, x \, y \to x \, a \, y, \mid a \in \mathbf{Anames}(D)\} \\
\qquad \cup \{\mathbf{new}^L \, x \to \mathtt{br} \, (x \, K) \, (\nu^L(x \, I)) \mid L \in \mathbf{Specs}(D)\} \\
\qquad \cup \{\mathbf{if}* \, x \, y \to \mathtt{br} \, x \, y, \\
\qquad\quad I \, x \, y \to x(y), \\
\qquad\quad K \, x \, y \to y, \\
\qquad\quad \star \to \mathtt{t}(\star)\}
\end{array}$$

Here, the types of terminals and non-terminals are given by:

$$\begin{array}{ll}
\star : \mathtt{o} & a, \mathtt{t}, \mathtt{call}, \nu^L : \mathtt{o} \to \mathtt{o} \\
I, K : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o} & \mathbf{if}*, \mathtt{br} : \mathtt{o} \to \mathtt{o} \to \mathtt{o} \\
\mathbf{new}^L : (((\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}) \to \mathtt{o}) \to \mathtt{o} & \\
\mathbf{acc}_a : ((\mathtt{o} \to \mathtt{o}) \to (\mathtt{o} \to \mathtt{o})) \to \mathtt{o} \to \mathtt{o} &
\end{array}$$

The transformation above follows the ideas sketched in Section 3.1, except the following points. For each function definition, an extra node $\mathtt{call}$ is inserted, to ensure that the value tree of $\mathcal{H}(D)$ does not contain $\bot$ (recall Remark 2.1). Unlike in Figures 3 and 4, the symbol $\star$ (which expresses a termination) is a *non-terminal*, and reduced to the infinite tree $\mathtt{t}(\mathtt{t}(\cdots))$. This is for ensuring that all the branches of the value tree of $\mathcal{H}(D)$ are infinite.

The following lemma guarantees that $\mathcal{H}(D)$ is certainly a higher-order recursion scheme.

LEMMA 3.2. *If D is well typed, then $\mathcal{H}(D)$ is a higher-order recursion scheme.*

Proof It suffices to show that the generated rewriting rules are well-typed. We define $\mathcal{H}(\tau)$, the translation of types into tree types, by: $\mathcal{H}(\mathbf{unit}) = \mathtt{o}$, $\mathcal{H}(\mathbf{R}) = (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$, and $\mathcal{H}(\tau_1 \to \tau_2) = \mathcal{H}(\tau_1) \to \mathcal{H}(\tau_2)$. Then, it is easy to see that $\Gamma \vdash e : \tau$ implies e is a term of type $\mathcal{H}(\tau)$ under the typing context $\mathcal{H}(\Gamma)$, and that all the rewriting rules (in the definition of $\mathcal{R}$ above) are well-typed. □

The tree automaton $\mathcal{M}(D)$ is constructed as follows. Let $\mathbf{Specs}(D) = \{L_1, \ldots, L_n\}$. Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_{S,1}, Q_{F,1})$, …, $M_n = (Q_n, \Sigma_n, \delta_n, q_{S,n}, Q_{F,n})$ be deterministic finite state automata (for strings) that accept $L_1, \ldots, L_n$ respectively. Here, $\Sigma_1 \cup \cdots \cup \Sigma_n \subseteq \mathbf{Anames}(D)$. We assume that the sets of states $Q_1, \ldots, Q_n$ are disjoint from each other. Then, $\mathcal{M}(D)$ is a Büchi

tree automaton $(Q, \Sigma, q_S, \Delta, Q_F)$ where:

$$\begin{array}{l} Q = Q_F = \{q_S, q_t\} \cup Q_1 \cup \cdots \cup Q_n \\ \Delta = \{(q_i, a, q_j) \mid \delta_k(q_i, a) = q_j \text{ for some } k\} \\ \quad \cup \{(q_S, \nu^{L_i}, q_{S,i}) \mid i \in \{1, \ldots, n\}\} \\ \quad \cup \{(q, \mathtt{t}, q_t) \mid q \in Q_{F,1} \cup \cdots \cup Q_{F,n} \cup \{q_t, q_S\}\} \\ \quad \cup \{(q, \nu^{L_i}, q_t) \mid q \in Q \setminus \{q_S\}, i \in \{1, \ldots, n\}\} \\ \quad \cup \{(q, \mathtt{br}, q, q) \mid q \in Q\} \\ \quad \cup \{(q, \mathtt{call}, q) \mid q \in Q\} \\ \quad \cup \{(q_t, a, q_t) \mid a \in \mathbf{Anames}(D)\} \end{array}$$

The 3rd set of Δ describes transitions for the program termination. The transitions are defined if the focused resource (i.e., the resource that has been instantiated to I) has been used up (i.e. if $q \in Q_{F,1} \cup \cdots \cup Q_{F,n}$), if no resource has been instantiated to I (i.e., if $q = q_S$), or if the automaton is already in the final state q_t. The 4th set of Δ describes transitions for the case where more than one resources have been instantiated to I (as in the subtree t_4 in Figure 4); in that case, the automaton goes to the final state q_t, so that all the remaining subtrees are ignored (by the last set of Δ). The 6th set of Δ says that `call` is just skipped (recall Remark 2.1).

EXAMPLE 3.1. *Recall the second program in Section 3.1:*

```
let rec f x y = if b then close(x);close(y)
                else read(x);write(y);f x y
in ...
```

It is expressed as the following program D:

$$\begin{array}{rcl} S & = & \mathbf{new}^{\mathtt{r}^*\mathtt{c}}\ C_1 \\ C_1\ x & = & \mathbf{new}^{\mathtt{w}^*\mathtt{c}}\ (C_2\ x) \\ C_2\ x\ y & = & F\ x\ y\ \star \\ F\ x\ y\ k & = & \mathbf{if}{*}\ (\mathbf{acc}_{\mathtt{c}}\ x\ (\mathbf{acc}_{\mathtt{c}}\ y\ k)) \\ & & \quad (\mathbf{acc}_{\mathtt{r}}\ x\ (\mathbf{acc}_{\mathtt{w}}\ y\ (F\ x\ y\ k))) \end{array}$$

The recursion scheme $\mathcal{H}(D)$ is $(\Sigma, \mathcal{N}, \mathcal{R}, S)$ where:

$$\begin{array}{l} dom(\Sigma) = \{\mathtt{r}, \mathtt{w}, \mathtt{c}, \nu^{\mathtt{r}^*\mathtt{c}}, \nu^{\mathtt{w}^*\mathtt{c}}, \mathtt{call}, \mathtt{t}, \mathtt{br}\} \\ dom(\mathcal{N}) = \{S, C_1, C_2, F, \mathbf{new}^{\mathtt{r}^*\mathtt{c}}, \mathbf{new}^{\mathtt{w}^*\mathtt{c}}, \mathbf{acc}_{\mathtt{r}}, \mathbf{acc}_{\mathtt{w}}, \mathbf{acc}_{\mathtt{c}}, \\ \qquad \mathbf{if}{*}, I, K, \star\} \\ \mathcal{R} = \{S \to \mathtt{call}(\mathbf{new}^{\mathtt{r}^*\mathtt{c}}\ C_1), \\ \qquad C_1\ x \to \mathtt{call}(\mathbf{new}^{\mathtt{w}^*\mathtt{c}}\ (C_2\ x)), \\ \qquad C_2\ x\ y \to \mathtt{call}(F\ x\ y\ \star), \\ \qquad F\ x\ y\ k \to \mathtt{call}(\mathbf{if}{*}\ (\mathbf{acc}_{\mathtt{c}}\ x\ (\mathbf{acc}_{\mathtt{c}}\ y\ k)) \\ \qquad\qquad (\mathbf{acc}_{\mathtt{r}}\ x\ (\mathbf{acc}_{\mathtt{w}}\ y\ (F\ x\ y\ k)))), \\ \qquad \mathbf{acc}_{\mathtt{r}}\ x\ y \to x\ \mathtt{r}\ y, \mathbf{acc}_{\mathtt{w}}\ x\ y \to x\ \mathtt{w}\ y, \\ \qquad \mathbf{acc}_{\mathtt{c}}\ x\ y \to x\ \mathtt{c}\ y, \\ \qquad \mathbf{new}^{\mathtt{r}^*\mathtt{c}}\ x \to \mathtt{br}\ (x\ K)\ (\nu^{\mathtt{r}^*\mathtt{c}}(x\ I)), \\ \qquad \mathbf{new}^{\mathtt{w}^*\mathtt{c}}\ x \to \mathtt{br}\ (x\ K)\ (\nu^{\mathtt{w}^*\mathtt{c}}(x\ I)), \cdots\} \end{array}$$

The tree automaton $\mathcal{M}(D)$ is shown in Figure 5. All the states are accepting states, so that a tree is rejected only if there is no run for the tree. For example, a tree of the form $\nu^{\mathtt{w}^*\mathtt{c}}(\mathtt{r}(\cdots))$ will be rejected. □

The correctness of the transformation is stated as follows.

THEOREM 3.3. *Let D be a (well-typed) program. Then, D is resource-safe if and only if $[\![\mathcal{H}(D)]\!]$ is accepted by $\mathcal{M}(D)$.*

To prove the theorem above, it suffices to establish a correspondence between the execution of D and the reduction of $\mathcal{H}(D)$. More specifically, (I) if $(\emptyset, S) \longrightarrow_D^* (H, e)$ and $H(x_i) = (w_i, L_i)$ for some $x_1, \ldots, x_k \in dom(H)$, then there exists t such that $S \longrightarrow_{\mathcal{H}(D)}^* t$ and $t^\perp$ has a path that matches a shuffle of $\nu^{L_1} w_1, \ldots, \nu^{L_k} w_k$ (with `br` and `call` being ignored); and (II) if $S \longrightarrow_{\mathcal{H}(D)}^* t$, then for every path π of $t^\perp$ that does not contain `t`, there exists H, e, and $x_1, \ldots, x_k \in dom(H)$ such that (i) $(\emptyset, S) \longrightarrow_D^* (H, e)$, (ii) π matches a shuffle of $\nu^{L_1} w_1, \ldots, \nu^{L_k} w_k$, and (iii) $H(x_i) = (w_i, L_i)$ for each $i \in \{1, \ldots, k\}$. See the extended version [25] for the proof.

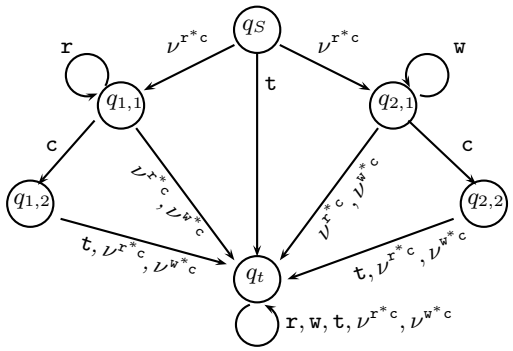

Figure 5. The transition diagram for the automaton $\mathcal{M}(D)$ in Example 3.1 (transitions for `br` and `call` are omitted)

The following corollary follows immediately from Theorems 2.1 and 3.3.

COROLLARY 3.4. *The resource usage verification problem is decidable.*

REMARK 3.3. *Instead of generating a single HORS and a tree automaton from a program, we can also generate a pair of an HORS and a tree automaton* for each occurrence of $\mathbf{new}^L$. *Then, the resource usage verification problem is reduced to* a set of *model-checking problems for HORS. For instance, the program in Example 3.1 can be transformed into the following two HORS's.*

$$\begin{array}{rcl} \mathcal{R}_1 & = & \{S \to \mathtt{call}(\mathbf{new}^{\mathtt{r}^*\mathtt{c}}\ C_1), \\ & & C_1\ x \to \mathtt{call}(C_2\ x\ K) \\ & & C_2\ x\ y \to \mathtt{call}(F\ x\ y\ k), \\ & & F\ x\ y\ k \to \mathtt{call}(\mathbf{if}{*}\ (\mathbf{acc}_{\mathtt{c}}\ x\ (\mathbf{acc}_{\mathtt{c}}\ y\ k)) \\ & & \qquad (\mathbf{acc}_{\mathtt{r}}\ x\ (\mathbf{acc}_{\mathtt{w}}\ y\ (F\ x\ y\ k)))), \ldots\} \\ \\ \mathcal{R}_2 & = & \{S \to \mathtt{call}(C_1\ K) \\ & & C_1\ x \to \mathtt{call}(\mathbf{new}^{\mathtt{w}^*\mathtt{c}}\ (C_2\ x)), \\ & & C_2\ x\ y \to \mathtt{call}(F\ x\ y\ k), \\ & & F\ x\ y\ k \to \mathtt{call}(\mathbf{if}{*}\ (\mathbf{acc}_{\mathtt{c}}\ x\ (\mathbf{acc}_{\mathtt{c}}\ y\ k)) \\ & & \qquad (\mathbf{acc}_{\mathtt{r}}\ x\ (\mathbf{acc}_{\mathtt{w}}\ y\ (F\ x\ y\ k)))), \ldots\} \end{array}$$

This alternative approach may be preferable in practice, as the size of each tree automaton is kept small.

4. Abstract Model Checking for Higher-Order Programs

We have so far considered the language having only resources and functions. This section informally discusses how to extend the verification method to deal with ordinary values (such as booleans and integers), by using the existing techniques for abstract model checking.

A naive approach would be to throw away all the values except resources and functions. That, however, leads to many false alarms. For example, consider the following program:

```
let f b x = (if b then lock(x) else ());
            (if b then unlock(x) else ())
in f true (newlock())
```

If `b` is true, then `f b x` locks and unlocks `x`; otherwise it does nothing on `x`. Therefore, it safely uses the lock `x`. If we ignore the parameter b, however, we get the following abstract program (in

CPS):

$$F\ k\ x = \mathbf{if}*\ (\mathbf{acc}_{\mathtt{lck}}\ x\ (C\ x\ k))\ (C\ x\ k)$$
$$C\ x\ k = \mathbf{if}*\ (\mathbf{acc}_{\mathtt{ulck}}\ x\ k)\ k$$
$$S = \mathbf{new}^{(\mathtt{lck}\ \mathtt{ulck})^*}\ (F\ \star)$$

Here, `lck` and `ulck` stand for lock and unlock operations respectively. The abstract program is not resource-safe, since it may unlock x without having locked x.

We can use techniques for abstract model checking to deal with the above problem. First, let us consider *boolean* higher-order programs, where we have only booleans, resources, and functions as values. The program given above is in fact a boolean program. It can be expressed in CPS as follows.

$$F\ b\ k\ x = \mathbf{if}\ b\ (\mathbf{acc}_{\mathtt{lck}}\ x\ (C\ b\ x\ k))\ (C\ b\ x\ k)$$
$$C\ b\ x\ k = \mathbf{if}\ b\ (\mathbf{acc}_{\mathtt{ulck}}\ x\ k)\ k$$
$$S = \mathbf{new}^{(\mathtt{lck}\ \mathtt{ulck})^*}\ (F\ 1\ \star)$$

Here, we express true and false by 1 and 0 respectively. We can easily eliminate the boolean parameters, and obtain:[5]

$$F_0\ k\ x = C_0\ x\ k \qquad F_1\ k\ x = \mathbf{acc}_{\mathtt{lck}}\ x\ (C_1\ x\ k)$$
$$C_0\ x\ k = k \qquad C_1\ x\ k = \mathbf{acc}_{\mathtt{ulck}}\ x\ k$$
$$S = \mathbf{new}^{(\mathtt{lck}\ \mathtt{ulck})^*}\ (F_1\ \star)$$

The resulting program can be verified by using the method in the previous section. Thus, the resource-safety verification problem remains decidable (Corollary 3.4) for the extension of the language in the previous section with booleans and conditionals. As the boolean elimination may cause the size of the program to blow up, it might be better to extend an HORS model checking algorithm (Ong's algorithm or the type-based algorithm discussed in Section 5) to deal with booleans directly.

If the language is extended with infinite value domains (such as integers and recursive data structures), then the resource usage verification problem becomes undecidable. For such an extension, we can use predicate abstractions together with counter-example-guided abstraction refinement [3]. An important point is that if an HORS fails to satisfy a property, a counter-example can be constructed by our model-checking algorithm for HORS's discussed in Section 5, as in the ordinary model checking for finite state systems. Therefore, the abstraction-refinement cycle can be organized in a standard manner.

The entire framework of verification is illustrated in Figure 1. A source program is first transformed into a system of top-level function definitions in CPS (Step 1 in Figure 1). Predicate abstraction is then applied to obtain a (non-deterministic) higher-order boolean program (Step 2). As in the standard abstract model checking with counter-example-guided abstraction refinement [3], we can start with the simplest abstraction (which ignores values completely), and then gradually refine abstractions. As explained above, boolean values are then eliminated and an abstract program of the language in Section 3.2 is obtained (Step 3). Then, the program is converted to a higher-order recursion scheme, as discussed in Section 3.3 (Step 4). The higher-order recursion scheme is then verified by using the type-based method described in Section 5 (Step 5). If the verification succeeds, we can conclude that the source program is resource-safe. Otherwise, we can construct a counterexample (which is a straight-line program without function calls) based on the verification algorithm for HORS described in Section 5. By investigating the corresponding execution sequence of the original program, we can check whether the counterexample is a real one (Step 6). For this purpose, the standard weakest precondition computation would suffice. If it is a real counterexample, then we can conclude that the program is unsafe. Otherwise, infer new predicates (by heuristics), and re-do predicate abstraction (Step 7). Repeat this cycle (which may not terminate) until the program is found to be safe or unsafe.

In the extended version of this paper [25], we demonstrate how the whole verification framework works by using an example.

5. Intersection Types for Higher-Order Recursion Schemes

This section discusses a verification method for higher-order recursion schemes (Step 5 in Figure 1). We restrict our attention to *safety properties* (the properties expressed by "trivial automata" [1]: recall Section 2.2). Note that the transformation described in Section 3 generates only safety properties.

One approach to verification of HORS would be to use Ong's algorithm based on game semantics [29]. In this section, however, we present an alternative, *type-based* method for verification of HORS. Advantages of our type-based method are as follows.

- Our type-based algorithm seems (arguably) much simpler than Ong's algorithm.
- The verification problem of HORS of order n is n-EXPTIME-complete [29], so that both our algorithm and Ong's algorithm suffer from the n-EXPTIME worst-case bottleneck.[6] Under the assumption that both the maximum size of the type of each non-terminal and that of the specification are bound by constants, however, the running time of the algorithm is *linear* in the size of the recursion scheme. On the other hand, it is not clear how to optimize Ong's algorithm so that it runs in linear time under the same assumption.
- By restricting the underlying type system, we can obtain a variety of approximate (i.e., incomplete) but more efficient verification algorithms for HORS. In fact, as discussed in Section 6, previous type-based methods for resource usage verification can be regarded as restrictions of the type system given in this section.
- HORS is "simply-typed"; however, our type-based verification method may be used for extensions of HORS with a limited form of polymorphic types.

Aehlig [1] has also proposed a model-checking algorithm for the same class of properties for HORS. His algorithm is closely related to ours, but less efficient than our algorithm (see Section 7).

5.1 Type System for HORS

Let $M = (Q_M, \Sigma_M, q_{M,S}, \Delta_M, Q_{M,F})$ be a Büchi tree automaton such that $Q_M = Q_{M,F}$. (Note that the Büchi tree automaton $\mathcal{M}(D)$ in Section 3 satisfies the condition $Q_M = Q_{M,F}$.) We omit the subscript M when it is clear from the context. We shall construct a type system for higher-order recursion schemes, such that an HORS $\mathcal{G}$ has type $q_{M,S}$ if and only if $[\![\mathcal{G}]\!]$ is accepted by M.

The idea is to refine the tree type $\mathtt{o}$ to an intersection type of the form $q_1 \wedge \cdots \wedge q_k$. Intuitively, q_i describes trees that are accepted by M from state q_i (i.e., accepted by $(Q_M, \Sigma_M, q_i, \Delta_M, Q_{M,F})$). $q_1 \wedge \cdots \wedge q_k$ denotes the intersection of the sets of trees accepted from the initial states $q_1, \ldots, q_k$. The types of function terms are also refined accordingly. The type $q_1 \to q_0$ describes functions that take a tree accepted from state q_1, and return a tree accepted from state q_0. For example, in Example 3.1, `c` has type $(q_{1,2} \to q_{1,1}) \wedge (q_{2,2} \to q_{2,1})$. The terminal `br` has type $\bigwedge\{q \to q \to q \mid$

[5] In general, we can express a function F of type $\mathbf{bool} \to \cdots \to \mathbf{bool} \to \tau$ (where τ is not of the form $\mathbf{bool} \to \tau'$) by a (finite) set of functions $F_{b_1 \cdots b_n}$, where $F_{b_1 \cdots b_n}$ behaves like $F(b_1, \ldots, b_n)$.

[6] Actually, our algorithm deals with only safety properties. We are currently investigating whether the worst-case complexity remains the same for the safety properties.

$q \in Q_M\}$. To avoid the confusion between types denoted by κ and intersection types, we shall call the types denoted by κ *kinds* below.

The set of "well-formed" intersection types is defined by the relations $\sigma :: \kappa$ and $\theta ::_a \kappa$, which should be read "σ is a type of kind κ," and "θ is an atomic type of kind κ," respectively. We exclude out ill-formed types like $q_1 \wedge (q_2 \to q_3)$.

DEFINITION 5.1 (intersection types). *The relations $\sigma :: \kappa$ and $\theta ::_a \kappa$ are the least relations closed under the following rules:*

$$\frac{}{q_i ::_a o} \qquad \frac{\sigma :: \kappa_1 \qquad \theta ::_a \kappa_2}{\sigma \to \theta ::_a \kappa_1 \to \kappa_2}$$

$$\frac{\theta_i ::_a \kappa \quad \text{for each } i \in \{1,\ldots,n\}}{\bigwedge\{\theta_1,\ldots,\theta_n\} :: \kappa}$$

Note that we exclude out types like $q_1 \to (q_2 \wedge q_3)$. That is for a technical convenience in discussing a type inference algorithm later. Note that $q_1 \to (q_2 \wedge q_3)$ can be replaced by $(q_1 \to q_2) \wedge (q_1 \to q_3)$ (according to the intuitions explained above). We often write $\theta_1 \wedge \cdots \wedge \theta_n$ for $\bigwedge\{\theta_1,\ldots,\theta_n\}$. We write $\top$ for $\bigwedge\{\,\}$. We identify q with $\bigwedge\{q\}$.

A type judgment for terms of HORS is of the form $\Theta \vdash_M t : \sigma$, where Θ, called a type environment, is a mapping from variables and defined function symbols to types. $\Theta \vdash_M t : \sigma$ is the least relation closed under the following rules.

$$\frac{\Theta \vdash_M t : \theta_i \text{ (for each } i \in \{1,\ldots,n\})}{\Theta \vdash_M t : \bigwedge\{\theta_1,\ldots,\theta_n\}}$$

$$\Theta, x : \theta_1 \wedge \cdots \wedge \theta_n \vdash_M x : \theta_j \ (j \in \{1,\ldots,n\})$$

$$\Theta, F : \theta_1 \wedge \cdots \wedge \theta_n \vdash_M F : \theta_j \ (j \in \{1,\ldots,n\})$$

$$\frac{(q, a, q_1, \ldots, q_n) \in \Delta_M}{\Theta \vdash_M a : q_1 \to \cdots \to q_n \to q}$$

$$\frac{\Theta \vdash_M t_1 : \sigma \to \theta \qquad \Theta \vdash_M t_2 : \sigma}{\Theta \vdash_M t_1 t_2 : \theta}$$

Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ be a higher-order recursion scheme where $\Sigma = \Sigma_M$. $(\mathcal{G}, t)$ is defined to have type σ, written $\vdash_M (\mathcal{G}, t) : \sigma$, if there exists Θ such that:

1. $dom(\Theta) = dom(\mathcal{N})$;
2. $\Theta(F) :: \mathcal{N}(F)$ for each $F \in dom(\Theta)$;
3. If $F\,\widetilde{x} \to e \in \mathcal{R}$ and $\Theta \vdash_M F : \widetilde{\sigma} \to q$, then $\Theta, \widetilde{x} : \widetilde{\sigma} \vdash_M e : q$ holds; and
4. $\Theta \vdash_M t : \sigma$.

EXAMPLE 5.1. *Recall the HORS in Example 3.1 (and its specification automaton in Figure 5). S has type q_S under the following assignment of types to C_1, C_2, and F.*

$$\begin{array}{l} C_1 : (\sigma_K \to q_S) \wedge (\sigma_I \to q_{1,1}) \\ C_2 : (\sigma_K \to \sigma_K \to q_S) \wedge (\sigma_I \to \sigma_K \to q_{1,1}) \\ \quad \wedge (\sigma_K \to \sigma_I \to q_{2,1}) \wedge (\sigma_I \to \sigma_I \to q_t) \\ F : (\sigma_K \to \sigma_K \to q_S \to q_S) \wedge (\sigma_I \to \sigma_K \to q_{1,2} \to q_{1,1}) \\ \quad \wedge (\sigma_K \to \sigma_I \to q_{2,2} \to q_{2,1}) \wedge (\sigma_I \to \sigma_I \to q_t \to q_t) \end{array}$$

where

$$\sigma_I = \bigwedge\{(q_i \to q_j) \to q_i \to q_j \mid q_i, q_j \in Q\}$$
$$\sigma_K = \bigwedge\{(q_i \to q_j) \to q_k \to q_k \mid q_i, q_j, q_k \in Q\}$$

5.2 Soundness and Completeness

We first prove the soundness of the type system.

THEOREM 5.1 (soundness). *Let $\mathcal{G}$ be an HORS and $M = (Q_M, \Sigma_M, q_{M,S}, \Delta_M, Q_M)$ be a Büchi tree automaton. If $\vdash_M (\mathcal{G}, S) : q_{M,S}$, then $[\![\mathcal{G}]\!]$ is accepted by M.*

The proof of the theorem above is similar to standard type soundness proofs. We first prove that typing is preserved by reductions.

LEMMA 5.2 (type preservation). *If $\vdash_M (\mathcal{G}, t) : \sigma$ and $t \longrightarrow_{\mathcal{G}} t'$, then $\vdash_M (\mathcal{G}, t') : \sigma$.*

Proof Straightforward case analysis on the rule used for deriving $t \longrightarrow_{\mathcal{G}} t'$. □

Next, we show that if $(\mathcal{G}, t)$ is well-typed, then the "concretized" part of t, i.e., the part of t that has been already evaluated to tree nodes, is accepted by the automaton. The "concretized" part of t is expressed by $t^\perp$ defined in Section 2. Here, we regard $\perp$ as the infinite tree $\perp(\perp(\cdots))$. For example, $\mathtt{r}(\mathtt{c}(Ft))^\perp$ is the infinite tree $\mathtt{r}(\mathtt{c}(\perp(\perp(\cdots)))))^\perp$. We write $M^\perp$ for the following automaton, obtained from M by adding a dummy transition for $\perp$:

$$(Q_M, \Sigma_M \cup \{\perp\}, q_{M,S}, \Delta \cup \{(q, \perp, q) \mid q \in Q_M\}, Q_M).$$

LEMMA 5.3. *If $\vdash_M (\mathcal{G}, t) : q$, then $t^\perp$ is accepted by $M^\perp$ from state q.*

Proof The proof proceeds by induction on the structure of $t^\perp$. If $t^\perp = \perp$, the result follows immediately. Otherwise, $t^\perp$ is of the form $a\ t'_1\ \cdots\ t'_n$. In this case, t is of the form $a\ t_1\ \cdots\ t_n$ with ${t_i}^\perp = t'_i$ for $i = 1,\ldots,n$. By $\vdash_M (\mathcal{G}, t) : q_S$, we have $\vdash_M a : q_1 \to \cdots \to q_n \to q$ and $\vdash_M (\mathcal{G}, t_i) : q_i$ $(i = 1,\ldots,n)$. By the induction hypothesis, ${t_i}^\perp$ is accepted by M from state q_i. $\vdash_M a : q_1 \to \cdots q_n \to q$ implies also that M has the transition $(q, a, q_1, \ldots, q_n)$. Therefore, $t^\perp$ must be accepted by M from state q. □

We are now ready to prove Theorem 5.1.

Proof of Theorem 5.1 The proof proceeds by contradiction. Suppose that $\vdash_M (\mathcal{G}, t) : q_S$ holds, but $[\![\mathcal{G}]\!]$ is *not* accepted, i.e., the run of M over $[\![\mathcal{G}]\!]$ gets stuck. Then, there must exist t such that $S \longrightarrow_{\mathcal{G}}^* t$ and the run of $M^\perp$ for $t^\perp$ gets stuck. By Lemma 5.2, $\vdash_M (\mathcal{G}, t) : q_S$. Thus, by Lemma 5.3, $t^\perp$ must be accepted by $M^\perp$; hence a contradiction. □

Next, we prove the completeness of the type system.

THEOREM 5.4 (completeness). *Let $\mathcal{G}$ be an HORS and $M = (Q_M, \Sigma_M, q_{M,S}, \Delta_M, Q_M)$ be a Büchi tree automaton. If $[\![\mathcal{G}]\!]$ is accepted by M, then $\vdash_M (\mathcal{G}, S) : q_{M,S}$.*

We prepare a few definitions and lemmas before proving the theorem. We write $[\![(\mathcal{G}, t)]\!]$ for the infinite term generated from t by the rewriting rules of $\mathcal{G}$. (In other words, for $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$, $[\![(\mathcal{G}, t)]\!]$ is $[\![\mathcal{G}']\!]$ where $\mathcal{G}' = (\Sigma, \mathcal{N} \uplus \{S'\}, \mathcal{R} \uplus \{S' \to t\}, S')$.)

We add the axiom $\Theta \vdash_M \perp : q$ (for any $q \in Q$) to the type system. We identify the infinite tree $\perp(\perp(\cdots))$ with the term $\perp$ below: When we are talking about typing, $\perp$ is interpreted as the term $\perp$, while when $\perp$ is an input to an automaton, it is interpreted as the infinite tree.

LEMMA 5.5. *If $t^\perp$ is accepted by $M^\perp$ from state q, then $\emptyset \vdash_M {t^\perp} : q$.*

Proof The proof proceeds by induction on the structure of $t^\perp$. If $t^\perp = \perp$, the result follows immediately. Otherwise, $t^\perp$ is of the form $a\ t'_1\ \cdots\ t'_n$. In this case, t is of the form $a\ t_1\ \cdots\ t_n$ with ${t_i}^\perp = t'_i$ for $i = 1,\ldots,n$. By the assumption that $t^\perp$ is accepted by $M^\perp$ from state q, there must be states $q_1,\ldots,q_n$ such

that (i) M has the transition $(q, a, q_1, \ldots, q_n)$, and (ii) $t'_1, \ldots, t'_n$ are accepted by $M^\perp$ from states $q_1, \ldots, q_n$ respectively. By the induction hypothesis and the condition (ii), we have $\emptyset \vdash_M t'_i : q_i$ for each i. The condition (i) implies that $\emptyset \vdash_M a : q_1 \to \cdots \to q_n \to q$. Thus, we have $\emptyset \vdash_M t^\perp : q$ as required. $\square$

The following lemma says that typing is preserved by the *inverse* of the reduction relation (c.f. Lemma 5.2).

LEMMA 5.6. *If* $\vdash_M (\mathcal{G}, t') : \sigma$ *and* $t \longrightarrow_{\mathcal{G}} t'$, *then* $\vdash_M (\mathcal{G}, t) : \sigma$.

Proof This follows by induction on the derivation of $t \longrightarrow_{\mathcal{G}} t'$. Since the induction step is trivial, we show only the base case, where $t = F\ \widetilde{s}$ and $t' = [\widetilde{s}/\widetilde{x}]t_0$. Here, $F\ \widetilde{x} \to t_0$ is a rewriting rule of $\mathcal{G}$. We can assume without loss of generality that σ is a type of the form q. Let Θ be the type environment witnessing $\vdash_M (\mathcal{G}, t') : \sigma$. Note that we have $\Theta \vdash_M t' : q$. Let $\{\theta_{i1}, \ldots, \theta_{ik_i}\}$ be the set of atomic types assigned to s_i in the type derivation for $\Theta \vdash_M t' : q$. Then, we can construct a derivation for $\Theta, x_1 : \bigwedge\{\theta_{11}, \ldots, \theta_{1k_1}\}, \ldots, x_n : \bigwedge\{\theta_{n1}, \ldots, \theta_{nk_n}\} \vdash_M t_0 : q$. Let Θ' be the type environment obtained from Θ by replacing the type of F with $\Theta(F) \wedge (\bigwedge\{\theta_{11}, \ldots, \theta_{1k_1}\} \to \cdots \to \bigwedge\{\theta_{n1}, \ldots, \theta_{nk_n}\} \to q)$. Then, we have $\Theta' \vdash_M t : q$, which implies $\vdash_M (\mathcal{G}, t) : q$. $\square$

We are now ready to prove Theorem 5.4.

Proof of Theorem 5.4 Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where $\mathcal{R} = \{F_1\ \widetilde{x}_1 \to t_1, \ldots, F_n\ \widetilde{x}_n \to t_n\}$. Let $\mathcal{TE}$ be the set of type environments:

$$\{\Theta \mid dom(\Theta) = \{F_1, \ldots, F_n\}, \Theta(F_i) :: \mathcal{N}(F_i) \text{ for each } i\}.$$

We define a mapping $\mathcal{F}$ from $\mathcal{TE}$ to $\mathcal{TE}$ by:

$$\mathcal{F}(\Theta)(F_i) = \bigwedge\{\widetilde{\sigma} \to q \mid \Theta, \widetilde{x}_i : \widetilde{\sigma} \vdash_M t_i : q\}$$

We write $\Theta_1 \sqsubseteq \Theta_2$ if $\Theta_1(F_i) = \bigwedge\{\theta_1, \ldots, \theta_m\}$ and $\Theta_2(F_i) = \bigwedge\{\theta_1, \ldots, \theta_n\}$ holds for each F_i with $\{\theta_1, \ldots, \theta_m\} \subseteq \{\theta_1, \ldots, \theta_n\}$. Note that $\mathcal{F}$ is monotonic with respect to $\sqsubseteq$.

Let $\Theta_{\mathbf{gfp}}$ be the greatest post-fixed point of $\mathcal{F}$, i.e., the greatest type environment such that $\Theta \sqsubseteq \mathcal{F}(\Theta)$. Since $\mathcal{TE}$ is a finite set, $\varphi_{\mathbf{gfp}} = \mathcal{F}^m(\Theta_0)$ for some m, where $\Theta_0(F_i) = \bigwedge\{\theta \mid \theta ::_a \mathcal{N}(F_i)\}$.

Then, $\vdash_M (\mathcal{G}, S) : q_S$ if and only if $\Theta_{\mathbf{gfp}} \vdash_M S : q_S$. (Note that the third condition in the definition of $\vdash_M (\mathcal{G}, t) : \sigma$ is equivalent to $\Theta \sqsubseteq \mathcal{F}(\Theta)$.)

Now, suppose that $[\![\mathcal{G}]\!]$ is accepted by M.

From $\mathcal{G}$, we construct a recursion-free HORS $\mathcal{G}^{(m)}$ as follows. For each defined function symbol F_i, prepare new defined function symbols $F_{i,0}, \ldots, F_{i,m}$ (where $S = S_m$). Then replace each rewriting rule $F_i\ \widetilde{x} \to t_i$ with the following rewriting rules:

$F_{i,k}\ \widetilde{x} \to [F_{1,k-1}/F_1, \ldots, F_{n,k-1}/F_n]t_i$ for $k \in \{1, \ldots, m\}$
$F_{i,0}\ \widetilde{x} \to \bot$

By the definition, $[\![\mathcal{G}^{(m)}]\!]$ is an approximation of $[\![\mathcal{G}]\!]$, i.e., $[\![\mathcal{G}^{(m)}]\!]$ is obtained by replacing some subtrees of $[\![\mathcal{G}]\!]$ with $\bot$. So, $[\![\mathcal{G}^{(m)}]\!]$ is accepted by $M^\perp$. Thus, by Lemma 5.5, we have $\vdash_M (\mathcal{G}^{(m)}, [\![\mathcal{G}^{(m)}]\!]) : q_S$. Because $\mathcal{G}^{(m)}$ is recursion-free, $S \longrightarrow^*_{\mathcal{G}^{(m)}} [\![\mathcal{G}^{(m)}]\!]$. Thus, By Lemma 5.6, $\vdash_M (\mathcal{G}^{(m)}, S) : q_S$. By the construction of $\mathcal{G}^{(m)}$, $\Theta_{\mathbf{gfp}} \vdash_M S : q_S$ (recall that $\Theta_{\mathbf{gfp}} = \mathcal{F}^m(\Theta_0)$), which implies $\vdash_M (\mathcal{G}, S) : q_S$. $\square$

5.3 Type Checking Algorithm

We discuss a type checking algorithm and its time complexity in this subsection.

The proof of Theorem 5.4 gives the following type checking algorithm:

1. Let Θ_0 be $F_1 : \bigwedge\{\theta \mid \theta ::_a \mathcal{N}(F_1)\}, \ldots, F_n : \bigwedge\{\theta \mid \theta ::_a \mathcal{N}(F_n)\}$.
2. Compute $\mathcal{F}^1(\Theta_0), \mathcal{F}^2(\Theta_0), \ldots$, and find m such that either $\mathcal{F}^m(\Theta_0) = \mathcal{F}^{m+1}(\Theta_0)$ or $\mathcal{F}^m(\Theta_0) \not\vdash_M S : q_S$.
3. Answer whether $\mathcal{F}^m(\Theta_0) \vdash_M S : q_S$ holds.

Note that the second step must terminate, as $\mathcal{F}^i(\Theta_0)$ ranges over a finite set, and decreases monotonically. m is bound by the maximum length of a decreasing sequence, which is $|\mathcal{N}(F_1)| + \cdots + |\mathcal{N}(F_n)|$.

When the type checking fails, a type error slice [13] of $\mathcal{G}^{(m)}$ serves as a counter-example, which will be given as an input for Step 6 in Figure 1.

We now discuss the time complexity of the above algorithm. Let $\mathcal{G}$ be a higher-order recursion scheme of order $N(\geq 1)$, and M be a Büchi tree automaton $M = (Q, \Sigma, q_S, \Delta, Q)$. Let $|\mathcal{G}|$ and $|Q|$ be the size of (the rules of) $\mathcal{G}$ and that of Q respectively. The number of atomic types of kind o is $|Q|$, and the number of atomic types of kind $\mathsf{o} \to \cdots \to \mathsf{o} \to \mathsf{o}$ is $2^{|Q|} \times \cdots \times 2^{|Q|} \times (|Q|)$. In general, if κ has order N and arity k, the number of atomic types of kind κ is bound by $\mathbf{exp}_N(O(k|Q|)))$, where $\mathbf{exp}_N(x)$ is defined by:

$$\mathbf{exp}_0(x) = x \qquad \mathbf{exp}_{i+1}(x) = 2^{\mathbf{exp}_i(x)}$$

Thus, m is bound by $(\mathbf{exp}_N(O(K|Q|))) \times n$, where K is the maximum arity of $F_1, \ldots, F_n$. The cost for computing $\mathcal{F}(\Theta)$ is also bound by $\mathbf{exp}_N(O(K|Q|)) \times |\mathcal{G}|$, so that the running time of the algorithm is bound by $\mathbf{exp}_N(O(|Q||\mathcal{G}|))$. (Note that K and n are bound by $|\mathcal{G}|$.) This time complexity is analogous to that of Ong's algorithm using game semantics [29].[7] Note, however, that our algorithm can deal with only safety properties, while Ong's algorithm can deal with arbitrary properties expressed by modal μ-calculus.

The extremely high time complexity above may be disappointing. Note, however, that the high time complexity is attributed to the explosion of the size of the set of intersection types. Let us assume that the size of the *kind* of each defined function symbol is bound by a constant (hence so are K and N above). Here, the size of κ is defined by $|\mathsf{o}| = 0$ and $|\kappa_1 \to \kappa_2| = |\kappa_1| + |\kappa_2| + 1$. Let us also assume that $|Q|$ is also bound by a constant. Then, for a given κ, the size of the set of intersection types of kind κ is bound by a constant. Therefore, in the above algorithm, both the number of iterations m and the cost for each iteration is $O(|\mathcal{G}|)$, so that the algorithm runs in time $O(|\mathcal{G}|^2)$. By using Rehof and Mogensen's method [30] for solving constraints in finite semi-lattices, we can further optimize it to obtain a linear time algorithm.

For the purpose of resource usage verification, the above assumption about the size of kinds seems to be reasonable. Note that the kinds of defined function symbols correspond to the simple types of function variables of source programs. In realistic programs, the type size does not necessarily increase with the increase of the program size. In fact, the boundedness of type size is often assumed in the context of type-based program analysis [15, 21]. The assumption on the number of states of M also seems reasonable, because the automaton M is determined solely by the kinds of resources used in the program.

5.4 Type Refinement – Towards Efficient Type-Based Model Checking of HORS

The most significant bottleneck of the verification framework in Figure 1 is probably the phase for model-checking HORS (Step

[7] Actually, Ong's time complexity result is with respect to the size of a modal μ-calculus formula. Since the size of the equivalent tree automaton can be exponential in the size of the formula, we should modify the above type system and type checking algorithm to deal with *alternating tree automata* [9]. That modification is easy: just change the types of terminal symbols. Thus, the above time complexity result should hold even when a property is given by a modal μ-calculus formula.

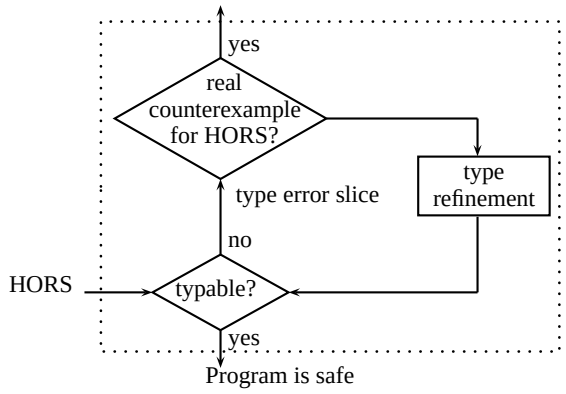

Figure 6. Type-Based Verification of HORS with Type Refinement. The dotted box corresponds to Step 5 in Figure 1

5). As discussed above, the complexity of the algorithm for order-n HORS is in general n-EXPTIME. Although the algorithm runs in time linear in the program size with the assumption on the boundedness of the type and specification sizes, the constant factor would be extremely large.

A possible solution to this problem would be to use an incomplete but efficient type system first, and then gradually refine the type system, as illustrated in Figure 6. Given an HORS $\mathcal{G}$ and an automaton M, use a simple type system first to check whether $\mathcal{G}$ has type q_S. If the type-checking succeeds, then the program is resource-safe. Otherwise, compute a *type error slice* [13]. Analyze the type error slice and check whether there is a real reduction sequence of $\mathcal{G}$ that violates the property. If so, output it as a possible counterexample for the original program (which is passed to Step 6 in Figure 1). Otherwise, refine the type system, and continue the cycle. Thus, we have now two refinement cycles in the verification framework in Figure 1; a cycle for predicate refinement, and another cycle for type refinement.

We have not yet studied exactly how types should be refined in the framework sketched above, but the intersection type system seems convenient for the type refinement framework above. By restricting the form of intersection types, we can obtain type systems of varying efficiency and expressive power. In fact, as we discuss in the next section, previous type systems for resource usage verification may be considered restricted forms of our intersection type system.

6. Comparison with Previous Type Systems for Resource Usage Verification

In this section, we discuss relationships between our intersection type system for HORS and three previous type systems for resource usage verification (or typestate): typestates [32], Igarashi and Kobayashi's type system [19], and flow-sensitive type qualifiers [12]. The discussion would provide a hint for designing the type refinement framework sketched in Section 5.4.

First, a straightforward consequence of Theorems 5.1 and 5.4 is that, for the language of Section 3, our method (the combination of the transformation in Section 3 and the type system in Section 5) is always at least as accurate as any method (including type systems, flow analysis, or whatever) for resource usage verification. That is also the case for languages with ordinary values (such as booleans and integers) for the three type systems [12, 19, 32], as they do not capture value-dependent information.

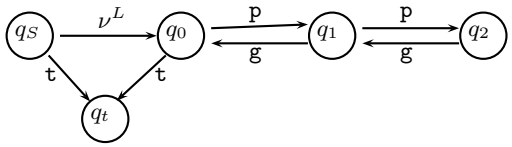

Figure 7. The specification automaton for 2-place buffer (unimportant transitions are omitted).

We use the program in Example 3.1 and the example below to compare our intersection type system and the three type systems in more detail.

EXAMPLE 6.1. *Consider the following program that accesses 2-place buffers.*

```
fun f x y = g x y; h x y;
fun g x y = put(x); put(y);
fun h x y = get(x); get(y);
if * then f (newbuf()) (newbuf())
else let x=newbuf() in f x x
```

Here, `newbuf()` *creates a new 2-place buffer, on which the operation* `put` *can be performed twice before* `get` *is performed.*

The program can be transformed into the following program D in our language.

$$\begin{array}{rcl} S & = & \mathbf{if}{*}\ (\mathbf{new}^L\ C_1)\ (\mathbf{new}^L\ C_2) \\ C_1\ x & = & \mathbf{new}^L\ (C_3\ x) \\ C_2\ x & = & F\ x\ x\ \star \\ C_3\ x\ y & = & F\ x\ y\ \star \\ F\ x\ y\ k & = & G\ x\ y\ (H\ x\ y\ k) \\ G\ x\ y\ k & = & \mathbf{acc}_{\mathtt{p}}\ x\ (\mathbf{acc}_{\mathtt{p}}\ y\ k) \\ H\ x\ y\ k & = & \mathbf{acc}_{\mathtt{g}}\ x\ (\mathbf{acc}_{\mathtt{g}}\ y\ k) \end{array}$$

Here, L is the set of sequences $s \in \{\mathtt{p}, \mathtt{g}\}$ such that (i) $\#_{\mathtt{g}}(s) = \#_{\mathtt{p}}(s)$, and (ii) for any prefix s' of s, $\#_{\mathtt{g}}(s') \leq \#_{\mathtt{p}}(s') \leq \#_{\mathtt{g}}(s') + 2$, where $\#_a(s)$ is the number of occurrences of a in s.

The specification automaton $\mathcal{M}(D)$ is shown in Figure 7.

The HORS $\mathcal{H}(D)$ has type q_S under the following assignment of types.

$$\begin{array}{l} F : \sigma \wedge (\sigma_I \to \sigma_I \to q_0 \to q_0) \wedge (\sigma_I \to \sigma_K \to q_0 \to q_0) \\ \quad \wedge (\sigma_K \to \sigma_I \to q_0 \to q_0) \wedge (\sigma_K \to \sigma_K \to q_S \to q_S) \\ G : \sigma \wedge (\sigma_I \to \sigma_I \to q_2 \to q_0) \wedge (\sigma_I \to \sigma_K \to q_1 \to q_0) \\ \quad \wedge (\sigma_K \to \sigma_I \to q_1 \to q_0) \wedge (\sigma_K \to \sigma_K \to q_S \to q_S) \\ H : \sigma \wedge (\sigma_I \to \sigma_I \to q_0 \to q_2) \wedge (\sigma_I \to \sigma_K \to q_0 \to q_1) \\ \quad \wedge (\sigma_K \to \sigma_I \to q_0 \to q_1) \wedge (\sigma_K \to \sigma_K \to q_S \to q_S) \end{array}$$

Here, σ_I and σ_K are the same as those in Example 5.1, and σ is $\sigma_I \to \sigma_I \to q_t \to q_t$. □

6.1 Typestate

Typestates [32] have been proposed as an extension of types with states, which determine what operations can be performed at each program point.[8] The states roughly correspond to the automaton states in this paper. Each program point is associated with a mapping from variables to typestates. A function's type is expressed as a pair of the type environments at the entry point and the exit point. For example, the function f in Example 3.1 is given type $x{:}\mathbf{R}(q_{1,1}), y{:}\mathbf{R}(q_{2,1})$ at the entry point, and $x{:}\mathbf{R}(q_{1,2}), y{:}\mathbf{R}(q_{2,2})$

[8] Actually, the original algorithm for checking typestates [32] is presented as a flow analysis. The following discussion is based on our interpretation of the original work from the viewpoint of type systems.

at the exit point. Here, $\mathbf{R}(q)$ is the type of a resource in state q. (We use states of the automaton in Figure 5.) Notice that this contains the same information as the following type of F given in Example 5.1:

$$F : \cdots \wedge (\sigma_I \to \sigma_K \to q_{1,2} \to q_{1,1}) \wedge (\sigma_K \to \sigma_I \to q_{2,2} \to q_{2,1})$$

The part $\sigma_I \to \sigma_K \to q_{1,2} \to q_{1,1}$ expresses the change of the state of the first resource, while $\sigma_K \to \sigma_I \to q_{2,2} \to q_{2,1}$ expresses the change of the state of the second resource. (Notice that since F is in the continuation-passing style, the types $q_{1,2}$ and $q_{2,2}$ of continuations describe the states at the exit point of f.) In general, a function that has typestates $x_1 : \mathbf{R}(q_1), x_2 : \mathbf{R}(q_2), \ldots, x_n : \mathbf{R}(q_n)$ at the entry point and $x_1 : \mathbf{R}(q_1'), x_2 : \mathbf{R}(q_2'), \ldots, x_n : \mathbf{R}(q_n')$ at the exit point will be given the following type in our type system:

$$\begin{array}{l}(\sigma_I \to \sigma_K \to \cdots \to \sigma_K \to q_1' \to q_1)\\ \wedge(\sigma_K \to \sigma_I \to \cdots \to \sigma_K \to q_2' \to q_2)\\ \wedge \cdots \wedge (\sigma_K \to \cdots \to \sigma_K \to \sigma_I \to q_n' \to q_n) \wedge \cdots\end{array}$$

A difference arises for the program in Example 6.1. f, g and h will be given the following typestates:

	entry point	exit point
f	$x : \mathbf{R}(q_0), y : \mathbf{R}(q_0)$	$x : \mathbf{R}(q_0), y : \mathbf{R}(q_0)$
g	$x : \mathbf{R}(q_0), y : \mathbf{R}(q_0)$	$x : \mathbf{R}(q_1), y : \mathbf{R}(q_1)$
h	$x : \mathbf{R}(q_1), y : \mathbf{R}(q_1)$	$x : \mathbf{R}(q_0), y : \mathbf{R}(q_0)$

In the typestates of [32], aliases are not allowed, so that x and y are assumed to be different resources. Therefore, the function call $f\ x\ x$ is not typable.

On the other hand, the HORS generated from the program is typable in our type system. The difference lies in the following parts of the types of F, G and H:

$$\begin{array}{l}F : (\sigma_I \to \sigma_I \to q_0 \to q_0) \wedge \cdots\\ G : (\sigma_I \to \sigma_I \to q_2 \to q_0) \wedge \cdots\\ H : (\sigma_I \to \sigma_I \to q_0 \to q_2) \wedge \cdots\end{array}$$

Those parts take care of the case where the two arguments are the same resource.

Another difference is polymorphism. In our type system, the term $\mathbf{acc}_{\mathtt{p}}\ y$ in the definition of G is given the following polymorphic type when y has type σ_I:

$$(q_1 \to q_0) \wedge (q_2 \to q_1)$$

On the other hand, in typestates, a unique type is assigned to each program point.

To summarize, the main differences between our type system and typestates are in the treatment of aliasing and polymorphism.

6.2 Igarashi and Kobayashi's type system

In Igarashi and Kobayashi's type system [19] (IK type system, for short), resource types are extended with a set of valid access sequences. For example, the function f in Example 3.1 is given the following type:

$$\mathbf{File}(\mathtt{r}^*\mathtt{c}) \to \mathbf{File}(\mathtt{w}^*\mathtt{c}) \to \mathbf{unit}$$

Here, the part $\mathbf{File}(\mathtt{r}^*\mathtt{c})$ means that the first argument will be read and then closed by the function. The set of access sequences $\mathtt{r}^*\mathtt{c}$ corresponds to the transition from $q_{1,1}$ to $q_{1,2}$ in the automaton. Thus, it expresses the same information as the part $\sigma_I \to \sigma_K \to q_{1,2} \to q_{1,1}$ of F's type in the intersection type system.

IK type system can deal with polymorphism and aliasing in a restricted manner. The function g in Example 6.1 is given the following type:

$$\mathbf{R}(\mathtt{p}) \to \mathbf{R}(\mathtt{p}) \to \mathbf{unit}.$$

Since the action $\mathtt{p}$ corresponds to the transition from q_1 to q_2 and also the one from q_2 to q_3, the first occurrence of $\mathbf{R}(\mathtt{p})$ subsumes the information expressed by:

$$(\sigma_I \to \sigma_K \to q_2 \to q_1) \wedge (\sigma_I \to \sigma_K \to q_3 \to q_2).$$

Moreover, IK type system does not require that the first and second arguments are different resources. Given the call $f\ x\ x$, IK type system assigns to x the type $\mathbf{R}(\mathtt{p} \otimes \mathtt{p})$. Here, $L_1 \otimes L_2$ represents the shuffle of the languages L_1 and L_2; In this case, $\mathtt{p} \otimes \mathtt{p}$ denotes the singleton set $\{\mathtt{pp}\}$. Thus, the above type expresses the same information as:

$$\begin{array}{l}G : (\sigma_I \to \sigma_I \to q_3 \to q_1) \wedge (\sigma_K \to \sigma_K \to q_1 \to q_1) \wedge\\ (\sigma_I \to \sigma_K \to q_2 \to q_1) \wedge (\sigma_K \to \sigma_I \to q_2 \to q_1) \wedge\\ (\sigma_I \to \sigma_K \to q_3 \to q_2) \wedge (\sigma_K \to \sigma_I \to q_3 \to q_2)\end{array}$$

Handling of aliasing and polymorphism in IK type system is, however, limited. For example, consider the following function:

$$F\ x\ y\ c = \mathbf{acc}_{\mathtt{r}}\ x\ (\mathbf{acc}_{\mathtt{c}}\ y\ c)$$

The intersection type system can assign to F the following type:

$$\cdots \wedge (\sigma_I \to \sigma_I \to q_2 \to q_1) \wedge \cdots,$$

which expresses the information that if x and y are the same resource, then the resource state will change from q_1 to q_2. On the other hand, IK type system assigns to (the direct-style counterpart of) F the following type:

$$\mathbf{File}(\mathtt{r}) \to \mathbf{File}(\mathtt{c}) \to \mathbf{unit}$$

Given the call $F\ x\ x$, x is given the type $\mathbf{File}(\mathtt{r} \otimes \mathtt{c})$. The type means that x will be either read and then closed, or closed and then read. Thus, $F\ x\ x$ will be rejected. IK type system also suffers from lack of enough polymorphism: If F_1 and F_2 are passed to the same function G (as in $G(F_1)$ and $G(F_2)$), the same type is assigned to F_1 and F_2.

6.3 Flow-Sensitive Type Qualifiers

Foster et al. [12] proposed a type-based analysis for checking typestates (or flow-sensitive type qualifiers in their terminology). A resource type is annotated with a *region*, like $\mathbf{File}(\rho)$. A region is an abstract set of concrete resources (or locations). The type system then keeps track of the state of each region. For example, the function f in Example 3.1 is given the following type:

$$\begin{array}{c}(\{\rho_1 \mapsto q_{1,1}, \rho_2 \mapsto q_{2,1}\}, \mathbf{File}(\rho_1) \times \mathbf{File}(\rho_2))\\ \to (\{\rho_1 \mapsto q_{1,2}, \rho_2 \mapsto q_{2,2}\}, \mathbf{unit})\end{array}$$

Here, the type means that the function takes a pair of resources in regions ρ_1 and ρ_2, and that the states of regions ρ_1 and ρ_2 are $q_{1,1}$ and $q_{2,1}$ before the function call and $q_{1,2}$ and $q_{2,2}$ after the call. Thus, region names serve as the same role as variables in typestates [32], for keeping track of resource states, and the relationship between our intersection type system and the type qualifiers is similar to that between ours and typestates, except the treatment of aliases through the special construct **restrict**.

As in typestates, the type qualifiers also suffer from the problem of aliases: Suppose that files x and y have different resources of the same type $\mathbf{File}(\rho)$, and that the state of ρ is $q_{1,1}$. Then, $\mathbf{close}(x)$ cannot change the state of ρ to $q_{1,2}$, because x is closed, but y is still opened.

To deal with the alias problem, Foster et al. [12] introduces a special construct called **restrict**. For example, consider the following program fragment:

```
let x=hd(l) in lock(x); unlock(x)
```

Here, `l` is a list of locks. Since all the elements of `l` are abstracted to the same region (say, ρ), the above program cannot be typed. It

can, however, be rewritten to:

```
let x=hd(l) in restrict^ρ' z=x in lock(z); unlock(z)
```

Here, the region for x is locally renamed to ρ' and the state of ρ' can be changed inside the body of `restrict`, as long as (i) the state of ρ' is changed back to the original state at the end of the body and (ii) the region ρ is not accessed in the body. A similar mechanism is also employed in the type system of Vault [8].

When a region ρ expresses more than one resources, the resource type $\mathbf{R}(\rho)$ corresponds to the union type $\sigma_I \vee \sigma_K$ in our type system. Given a resource of type $\sigma_I \vee \sigma_K$, we do not know whether we should keep track of the resource state. Thus, the reasoning for `restrict` above seems to be related to the following rule for union types:

$$\frac{\Theta, x:\sigma_I, k:q_i \vdash t:q_j \qquad \Theta, x:\sigma_K, k:q_i \vdash t:q_j}{\Theta, x:\sigma_I \vee \sigma_K, k:q_i \vdash t:q_j}$$

Here, k denotes the continuation. The reasoning for `restrict` seems to be a special case of the above rule, where $q_i = q_j$ and t does not access resources in the same region as x. In that case, the assumption $\Theta, x:\sigma_K, k:q_i \vdash t:q_i$ follows immediately, so that it suffices to check $\Theta, x:\sigma_I, k:q_i \vdash t:q_i$.

7. Related Work

HORS model checking The present work owes much to the theoretical studies of the model-checking problem for higher-order recursion schemes [2, 14, 22–24, 29]. The modal μ-calculus model-checking problem for HORS has been extensively studied recently. Knapik [22] showed that the problem is decidable for order-2 safe higher-order recursion schemes (where "safety" is a certain syntactic condition), and later extended the result to safe HORS of any order [23]. Knapik et al. [24] and Aehlig et al. [2] then independently showed that the model checking problem is decidable for order-2 HORS, without the safety assumption. Finally, Ong [29] has shown that the problem is decidable for HORS of arbitrary order. Their algorithms are based on automata and game theories and are rather involved. It would be interesting that, for the restricted fragment of modal μ-calculus, the same problem can be solved in a rather simple manner by using types. Aehlig [1] has also proposed a model-checking algorithm of HORS for the same class of safety properties as ours. His algorithm is much closer to our algorithm than Ong's algorithm, although Aehlig assigns set-theoretic functions to terms instead of our intersection types. His algorithm guesses an assignment of set-theoretic functions to terms, and then checks the correctness of the guess. Thus, his algorithm is less efficient than our algorithm; for example, for order-1 recursion scheme, their algorithm is double exponential in the number of states, while our algorithm is exponential.

As far as the author knows, applications of these decidability results to program verification have been limited so far, except some work on verification of higher-order pushdown systems [14].

Software model checking Thanks to the advance of abstract model-checking techniques (such as counter-example-guided abstraction refinement and lazy abstractions), model checking has become a popular technique for software verification [3–5, 16]. The existing model checkers are mainly targeted for imperative languages with first-order procedures. The treatment of higher-order functions is limited; for example, in SLAM [3], a call to function pointer in a C program is replaced by a non-deterministic choice of all the functions that it may point to. Thus, information is lost in this pre-processing phase for software model checking.

A trick similar to the one used in Section 3 (of instantiating a new resource to I or K non-deterministically) is used also in software model checking [7].

Resource usage verification A number of type-based or flow-based techniques for verification of temporal properties have been proposed, under various names (resource usage verification, typestate checking, etc.) [10–12, 19, 20, 26, 32]. Unlike our intersection type system, they are incomplete for higher-order programs. Since our intersection type system is complete (for value-free programs), it may be used as a good device for comparing different techniques; three of them have been already discussed in Section 6.

Some of the verification techniques [10, 11, 26] can analyze value-dependent information. In our verification framework (Figure 1), the value-dependent information is handled in Step 2, a separate phase from HORS model checking.

Model checking vs type systems Naik and Palsberg [27, 28] studied type systems equivalent to model checkers for an imperative language and an interrupt calculus. Their type systems and ours have some similarity: a state or a value is represented by an atomic type, and the effect of a statement is expressed by an intersection of function types (each of which represents a state transition). A major difference is that they consider only types of order 1, while we consider types of higher-orders to deal with higher-order functions. Naik and Palsberg [27, 28] uses union types also, while we do not use them. That is because we consider only *deterministic* higher-order recursion schemes. Union types may be useful for verification of a non-deterministic HORS (which generate a set of trees).

In the context of the π-calculus, there is an approach to combining types and model checking [6, 18]. In that approach, a type-based analysis is used to extract abstract programs, and then the extracted programs are model-checked. Information is lost in the type inference phase, which causes false alarms. On the other hand, our intersection type system is complete; Information is lost only in the phase for predicate abstractions (Step 2 in Figure 1).

Dependent types Dependent types have been a popular method for semi-automatic verification of higher-order programs. Heuristic techniques for automated inference of dependent types have been studied recently [31, 34], but their applicability seems to be still limited. Our verification framework in Figure 1 provides an alternative approach to inference of dependent types. A higher-order program with assertions can be expressed in our language, by using a single global resource and a single action `fail` to express an assertion violation. Using the framework in Figure 1, we can start with simple types, and then gradually refine dependent types until the whole program is type-checked.

Tree types Connections between types and tree automata have been studied in the context of languages for XML processing [17]. They deal with *finite* trees, while our type system deals with infinite trees.

8. Conclusion

We have proposed a novel framework for verification of temporal properties of higher-order programs, based on the recent result on the decidability of HORS model checking. There are two main contributions in this work. The first one is the reduction of the resource usage verification problem to the HORS model checking problem. As far as the author knows, this is the first practical application of the decidability of HORS model checking to program verification. The reduction enables a smooth integration of the techniques for abstract model checking (in particular, counter-example-guided abstraction refinement) into verification of higher-order programs.

The second contribution is a type-based algorithm for HORS model checking. Although only a fragment of the modal μ-calculus

is handled, our algorithm and its correctness proof seem to be significantly simpler than the previous algorithm [29] (for the full modal μ-calculus). The new type system also serves as a good device for comparing previous type systems for verification of temporal properties.

A lot of work is left for future work. On the practical side, the two refinement cycles in Figures 1 and 6 should be substantiated. A verification tool should be implemented to test the feasibility of the verification method. Extending the method to deal with recursive data structures and pointers would also be important. On the theoretical side, it would be interesting to find a type system for the full modal μ-calculus model checking of HORS.

Acknowledgment

I would like to thank Luke Ong for introducing me to higher-order recursion schemes and providing useful comments, Sriram Rajamani for references on model checking, Jens Palsberg for references on the work on types vs model checking. We would also like to thank anonymous referees and members of our research group for useful comments. This work was partially supported by Kakenhi 20240001.

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] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In *TLCA 2005*, volume 3461 of *LNCS*, pages 39–54. Springer-Verlag, 2005.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 2001*, pages 203–213, 2001.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[6] S. Chaki, S. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Proc. of POPL*, pages 45–57, 2002.

[7] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *Proc. of POPL*, pages 265–276, 2007.

[8] R. DeLine and M. Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *Proc. of PLDI*, 2002.

[9] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS 1991*, pages 368–377, 1991.

[10] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. *Sci. Comput. Program.*, 58(1-2):57–82, 2005.

[11] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[12] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. of PLDI*, pages 1–12, 2002.

[13] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

[14] M. Hague and C.-H. L. Ong. Symbolic backwards-reachability analysis for higher-order pushdown systems. In *FoSSaCS 2007*, volume 4423 of *LNCS*, pages 213–227. Springer-Verlag, 2007.

[15] N. Heintze and D. A. McAllester. Linear-time subtransitive control flow analysis. In *Proc. of PLDI*, pages 261–272, 1997.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL*, pages 58–70, 2002.

[17] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[18] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[19] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. Prog. Lang. Syst.*, 27(2):264–313, 2005. Preliminary summary appeared in Proceedings of POPL 2002.

[20] F. Iwama, A. Igarashi, and N. Kobayashi. Resource usage analysis for a functional language with exceptions. In *Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM 2006)*, pages 38–47. ACM Press, 2006.

[21] D. Kikuchi and N. Kobayashi. Type-based verification of correspondence assertions for communication protocols. In *Proceedings of APLAS 2007*, volume 4807 of *LNCS*, pages 191–205. Springer-Verlag, 2007.

[22] T. Knapik, D. Niwinski, and P. Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In *TLCA 2001*, volume 2044 of *LNCS*, pages 253–267. Springer-Verlag, 2001.

[23] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS 2002*, volume 2303 of *LNCS*, pages 205–222. Springer-Verlag, 2002.

[24] T. Knapik, D. Niwinski, P. Urzyczyn, and I. Walukiewicz. Unsafe grammars and panic automata. In *ICALP 2005*, volume 3580 of *LNCS*, pages 1450–1461. Springer-Verlag, 2005.

[25] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. An extended version, available from `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/hors.pdf`, 2008.

[26] P. Lam, V. Kuncak, and M. C. Rinard. Generalized typestate checking for data structure consistency. In *VMCAI 2005*, volume 3385 of *LNCS*, pages 430–447. Springer-Verlag, 2005.

[27] M. Naik. A type system equivalent to a model checker. Master Thesis, Purdue University.

[28] M. Naik and J. Palsberg. A type system equivalent to a model checker. In *ESOP 2005*, volume 3444 of *LNCS*, pages 374–388. Springer-Verlag, 2005.

[29] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS 2006*, pages 81–90. IEEE Computer Society Press, 2006.

[30] J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2):191–221, 1999.

[31] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI 2008*, pages 159–169, 2008.

[32] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *Transactions on Software Engineering*, 12(1):157–171, 1986.

[33] W. Thomas. Languages, automata, and logic. In *Handbook of formal languages, vol. 3*, pages 389–455, 1997.

[34] H. Unno and N. Kobayashi. On-demand refinement of dependent types. In *Proceedings of FLOPS 2008*, volume 4989 of *LNCS*, pages 81–96. Springer-Verlag, 2008.