Model-Checking Higher-Order Functions

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

Abstract

We propose a novel type-based model checking algorithm for higher-order recursion schemes. As shown by Kobayashi, verification problems of higher-order functional programs can easily be translated into model checking problems of recursion schemes. Thus, the model checking algorithm serves as a basis for verification of higher-order functional programs. To our knowledge, this is the first practical algorithm for model checking recursion schemes: all the previous algorithms *always* suffer from the n-EXPTIME bottleneck, not only in the worst case, and there was no implementation of the algorithms. We have implemented a model checker for recursion schemes based on the proposed algorithm, and applied it to verification of functional programs, including reachability, flow analysis and resource usage verification problems. According to our experiments, the model checker is surprisingly fast: it could automatically verify a number of small but tricky higher-order functional programs in less than a second.

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

We propose a new model-checking algorithm for higher-order recursion schemes. A higher-order recursion scheme (recursion scheme, for short) is a grammar for generating a (possibly infinite) tree. It is known that the modal μ-calculus model checking of recursion schemes (i.e. the problem "given a recursion scheme $\mathcal{G}$ and a modal μ-calculus formula ψ, does the tree generated by $\mathcal{G}$ satisfy ψ?") is decidable [24]. Kobayashi [18] has recently shown that verification problems of higher-order functional programs can be reduced to model checking problems of recursion schemes. The idea was to transform a functional program into a recursion scheme that generates all the possible event sequences of the program, so that the temporal properties of the program can be verified by model-checking the recursion scheme. Thus, the model-checking algorithm for recursion schemes can be used as a verification algorithm for higher-order functional programs.

The verification method based on recursion schemes has a number of attractive features. First, the verification method can be fully automated. Secondly, unlike other automated verification/analysis methods for functional programs, the method is sound and *complete* for an extension of the simply-typed λ-calculus with recursion and finite data domains (such as booleans): in fact, Kobayashi [18] developed a sound and complete algorithm for the resource usage verification problem (i.e. the problem "given a program of the simply-typed λ-calculus with recursion and resource creation/access primitives, decide whether the program accesses each resource according to the resource specification"; other verification problems such as the reachability and the flow analysis can be regarded as instances of the resource usage verification) [12]. Thirdly, the verification method can be nicely integrated with software model checking techniques, such as predicate abstractions and CEGAR (counter-example-guided abstraction refinement), as outlined in [18]. Thus, for the verification of higher-order programs, the model-checking of recursion schemes can play the same role as finite-state model checking for verification of while-programs, and pushdown model checking for verification of programs with first-order, recursive procedures: See Table 1.

A major obstacle in applying recursion schemes to program verification has been its high time complexity. Ong [24] has shown that the model checking problem of order-n recursion schemes (where n is, roughly, the same as the largest order of types in functional programs; see Section 2 for a precise definition) is n-EXPTIME complete. Although Kobayashi [18] has shown that, for a fragment of the modal μ-calculus (describing safety properties), the problem is linear under the assumption that the sizes of types and properties are bounded above by a constant (which means that n is also a constant), the constant factor is *huge*: it is n-exponential in the sizes of arities (of higher-order functions) and properties. Furthermore, all the known model-checking algorithms, including Ong's algorithm [24] and Kobayashi's one [18], suffer from this n-exponential time bottleneck not only in the worst case, but also in the *best case*. Thus, despite wide theoretical interests in the model checking of recursion schemes [2, 3, 11, 15, 16, 24], there has been no implementation of a model-checker for recursion schemes, to our knowledge.

To remedy the problem above, we propose a novel algorithm for model-checking recursion schemes, which is expected to run much faster for realistic inputs than the previous algorithms.[1] The new algorithm works only for the properties described by a certain subclass of the modal μ-calculus, but the subclass is sufficient for the verification of temporal safety properties of functional programs.[2] A basic idea behind the new algorithm is as follows. Kobayashi's previous algorithm [18] is based on intersection types, and the reason for the n-exponential blow-up of the complexity of

[1] Readers who are not familiar with recursion schemes may wish to consult Section 2 before reading the following paragraphs.

[2] Our algorithm can actually be extended to deal with the full modal μ-calculus, although it is unclear whether the extended algorithm has reasonable efficiency.

Programs	Models
while-programs	finite-state machines
procedural programs	pushdown systems
higher-order programs	higher-order recursion schemes

Table 1. Programs and models

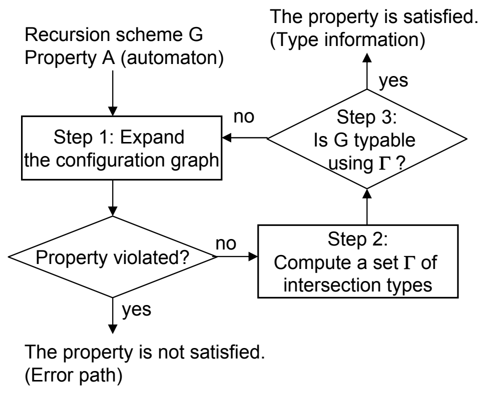


Figure 1. The overview of the verification algorithm

model checking recursion schemes is that the number of intersection types blows up exponentially with an increase of the order by 1. Kobayashi's algorithm tries to find a type derivation for a recursion scheme from such a huge set of intersection types (by a fixed-point computation). The key observation behind our new algorithm is that, in typical programs (or recursion schemes), we can expect that higher-order functions behave in a "uniform" manner, so that the number of intersection types that are actually assigned to the functions should be much smaller. Thus, given a recursion scheme, if we can effectively find candidates of the intersection types that are actually required for typing the recursion scheme, then the recursion scheme can be type-checked more efficiently.

To find candidates of intersection types, we extract an idea from the proof of the completeness of Kobayashi and Ong's type system for recursion schemes [20] (that is equivalent to the modal μ-calculus model checking of recursion schemes in the sense that the tree generated by a recursion scheme satisfies a given property if, and only if, the recursion scheme is well-typed). The proof of the completeness gives a kind of procedure to compute the intersection types of each function from an "infinite" configuration tree of a recursion scheme. (Here, the configuration tree shows how the recursion scheme is reduced.) As the configuration tree is infinite (and non-regular), the "procedure" is not actually an algorithm. We can, however, modify it to obtain an algorithm to compute candidates of intersection types from a *finite* subtree of the infinite configuration tree. Moreover, given a sufficiently large finite subtree, the set of candidates generated by the algorithm contains all the intersection types needed for typing the recursion scheme.

The overall structure of the algorithm is shown in Figure 1. First, (the initial symbol of) the recursion scheme is reduced a finite number of steps and a configuration graph is constructed (Step 1 in Figure 1). If a property violation is found during the reduction, then an error path is reported. Otherwise, a set Γ of type bindings is computed from the configuration graph (Step 2). As mentioned above, such an algorithm is obtained by modifying the "procedure" described in the completeness proof of Kobayashi and Ong's type system [20]. In Step 3, it is checked whether the recursion scheme is typed by using only the intersection types in Γ. Such a type-checking algorithm is obtained by modifying Kobayashi's fixed-point-based type checking algorithm [18]. If the recursion scheme is well-typed, then the property is satisfied (by the soundness of the type system [18]). Otherwise, go back to Step 1 and expand the configuration graph to get a larger set of intersection types. As we show later, the procedure eventually terminates and either proves or disproves the property. In the former case, a type environment for the recursion scheme is output as a certificate of the property satisfaction, while in the latter case, an error path is output as a witness of the property violation.

We have implemented a model checker for recursion schemes based on the algorithm described above. To our knowledge, this is the first implementation of a model checker for recursion schemes. We have tested it for a number of small but tricky examples, many of which have been obtained from verification problems of functional programs by using Kobayashi's translation [18]. According to the experimental results, the model checker is surprisingly fast, considering that the model checking problem is $(n-1)$-EXPTIME complete in general.[3]

The rest of this paper is structured as follows. Section 2 reviews the definition of recursion schemes and our previous work [18], in particular, the type system for recursion schemes and the translation of the resource usage verification problem into a model checking problem for recursion schemes. Section 3 describes the model checking algorithm and proves its correctness. Section 4 reports implementation and experimental results. Section 5 discusses a limitation of the current model checking algorithm. Section 6 discusses related work and Section 7 concludes.

2. Preliminaries

This section reviews the definition of higher-order recursion schemes and its relationship with program verification.

2.1 Higher-Order Recursion Schemes

A higher-order recursion scheme (recursion scheme, for short) is a grammar for generating a (possibly infinite) tree. From a programming language point of view, a recursion scheme is a term of the simply-typed λ-calculus with recursion and tree constructors (but without destructors).

We define the set of *sorts*, ranged over by κ, by:

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

The sort o describes trees. The sort $\kappa_1 \to \kappa_2$ describes a function that takes a value of sort κ_1 and returns a value of κ_2. The *order* and *arity* of κ, written $\mathit{order}(\kappa)$ and $\mathit{arity}(\kappa)$ respectively, are defined by:

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

A (deterministic) higher-order recursion scheme is a quadruple $(\Sigma, \mathcal{N}, \mathcal{R}, S)$, where

- Σ is a *ranked alphabet* i.e. a map from a finite set of symbols called *terminals* to sorts of order 0 or 1. We write $\mathit{arity}(a)$ for $\mathit{arity}(\Sigma(a))$.
- $\mathcal{N}$ is a map from a finite set of symbols called *non-terminals* to sorts. We require that $\mathcal{N}(S) = \mathsf{o}$.
- $\mathcal{R}$ is a map from the set of non-terminals (i.e. $\mathit{dom}(\mathcal{N})$) to terms of the form $\lambda\widetilde{x}.t$, where $\widetilde{x}$ abbreviates a sequence of variables, and t is a term of sort o. Here, the set of terms is defined

[3] The modal μ-calculus model checking of recursion schemes is n-EXPTIME, but as we discuss in Section 2, the model checking is $(n-1)$-EXPTIME complete for the class of properties considered in this paper.

as follows. A symbol (i.e., a terminal, non-terminal, or variable) of sort κ is a term of sort κ. If terms t_1 and t_2 have sorts $\kappa_1 \to \kappa_2$ and κ_1 respectively, then $t_1\ t_2$ is a term of sort κ_2. If $\mathcal{R}(F) = \lambda\widetilde{x}.t$, then $F\,\widetilde{x}$ must be a term of sort $\mathtt{o}$. (Thus, the sorts of the variables $\widetilde{x}$ are uniquely determined from that of F.)

- S is a non-terminal called the *start symbol*.

The *order* of a recursion scheme is the highest order of its non-terminals.

For a recursion scheme $\mathcal{G}$, the rewriting relation $\longrightarrow_{\mathcal{G}}$ is defined inductively by:

- $F\,\widetilde{s} \longrightarrow_{\mathcal{G}} [\widetilde{s}/\widetilde{x}]t$ if $\mathcal{R}(F) = \lambda\widetilde{x}.t$.
- If $t \longrightarrow_{\mathcal{G}} t'$, then $t\,s \longrightarrow_{\mathcal{G}} t'\,s$ and $s\,t \longrightarrow_{\mathcal{G}} s\,t'$.

Here, $[\widetilde{s}/\widetilde{x}]t$ is the term obtained by replacing variables $\widetilde{x}$ in t with terms $\widetilde{s}$. We omit the subscript $\mathcal{G}$ whenever it is clear from the context.

The *value tree* of $\mathcal{G}$, written $[\![\mathcal{G}]\!]$, is the (possibly infinite) $(dom(\Sigma) \cup \{\bot\})$-labeled tree obtained by infinitary, fair rewriting of the start symbol S. More precisely, given a set L of labels, an *L-labeled tree* T is a partial map from $\{1, \dots, m\}^*$ (where m is a natural number) to L such that (i) the domain of T is closed under the prefix operation, and (ii) if $\pi j \in dom(T)$, then $\pi i \in dom(T)$ for every $i \in \{1, \dots, j\}$. Given a term t, we write $t^{\bot}$ for the finite tree inductively defined by (i) $(F s_1 \cdots s_n)^{\bot} = \bot$ and (ii) $(a s_1 \cdots s_n)^{\bot} = a(s_1{}^{\bot}) \cdots (s_n{}^{\bot})$ (where $n \geq 0$). The value tree $[\![\mathcal{G}]\!]$ is defined as $\bigsqcup\{t^{\bot} \mid S \longrightarrow^*_{\mathcal{G}} t\}$, where $\bigsqcup_i T_i$ is defined by $(\bigsqcup_i T_i)(\pi) = \bigsqcup_i (T_i(\pi))$ for every $\pi \in \{1, \dots, A\}^*$ (where A is the largest arity), with $\bot \sqcup a = a$ for every $a \in dom(\Sigma)$. (Note that $\bigsqcup\{t^{\bot} \mid S \longrightarrow^*_{\mathcal{G}} t\}$ is well-defined, as there is exactly one rewriting rule for each non-terminal.)

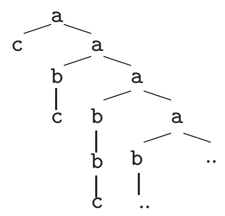
In the rest of this paper, we consider only recursion schemes whose value trees do not contain $\bot$. This condition is always satisfied by the recursion schemes generated by Kobayashi's translation of functional programs [18]. In general, given a recursion scheme $\mathcal{G}$ and a formula ψ, one can always transform them into $\mathcal{G}'$ and ψ' such that (i) $[\![\mathcal{G}']\!]$ satisfies ψ' if and only if $[\![\mathcal{G}]\!]$ satisfies ψ, and (ii) $[\![\mathcal{G}']\!]$ does not contain $\bot$: see [24].

The following example is taken from [20]. As the example indicates, the value tree of a recursion scheme may not be regular.

EXAMPLE 2.1. Consider the recursion scheme $\mathcal{G}_0 = (\Sigma, \mathcal{N}, \mathcal{R}, S)$, where: $\Sigma = \{\mathtt{a}{:}\mathtt{o} \to \mathtt{o} \to \mathtt{o}, \mathtt{b}{:}\mathtt{o} \to \mathtt{o}, \mathtt{c}{:}\mathtt{o}\}, \mathcal{N} = \{S{:}\mathtt{o}, F{:}\mathtt{o} \to \mathtt{o}\}$, and $\mathcal{R} = \{S \to F\,\mathtt{c}, F \to \lambda x.\mathtt{a}\,x\,(F(\mathtt{b}\,x))\}$. S is reduced as follows.

$$
\begin{aligned}
S &\longrightarrow F\,\mathtt{c} \longrightarrow \mathtt{a}\,\mathtt{c}\,(F(\mathtt{b}\,\mathtt{c})) \\
&\longrightarrow \mathtt{a}\,\mathtt{c}\,(\mathtt{a}\,(\mathtt{b}\,\mathtt{c})\,(F(\mathtt{b}(\mathtt{b}\,\mathtt{c})))) \longrightarrow \dots
\end{aligned}
$$

The value tree $[\![\mathcal{G}_0]\!]$ is shown as follows.



Ong [24] showed the decidability of the modal μ-calculus model checking of recursion schemes.

THEOREM 2.1 (Ong [24]). *The model checking problem: "Given a recursion scheme $\mathcal{G}$ of order n, and a modal μ-calculus formula ψ, does $[\![\mathcal{G}]\!]$ satisfy ψ?" is n-EXPTIME-complete.*

In this paper, we consider only a subset of the model checking problem, where the properties are restricted to those described by deterministic Büchi automata with a trivial acceptance condition (where all the states are final).

DEFINITION 2.1. A *deterministic Büchi automaton with a trivial acceptance condition* (a *deterministic trivial automaton*, for short) $\mathcal{B}$ is a quadruple:

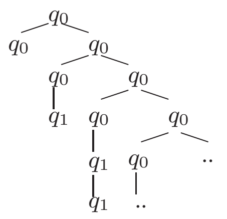
$$(\Sigma, Q, \delta, q_0)$$

where Σ is a ranked alphabet, Q is a set of states, δ, called a transition function, is a partial map from $Q \times dom(\Sigma)$ to Q^* such that if $\delta(q, a) = q_1 \cdots q_k$, then $k = arity(a)$. A $dom(\Sigma)$-labeled tree T is *accepted* by $\mathcal{B}$ if there is a Q-labeled tree R such that (i) $dom(T) = dom(R)$; (ii) for every $x \in dom(R)$, $\delta(R(x), T(x)) = R(x1) \cdots R(xm)$ where $m = arity(T(x))$. R is called a *run tree* of $\mathcal{B}$ over T.

EXAMPLE 2.2. *Consider the automaton $\mathcal{B}_0 := (\Sigma, \{q_0, q_1\}, \delta, q_0)$ where*

$$
\begin{aligned}
&\Sigma = \{\mathtt{a} \mapsto \mathtt{o} \to \mathtt{o} \to \mathtt{o}, \mathtt{b} \mapsto \mathtt{o} \to \mathtt{o}, \mathtt{c} \mapsto \mathtt{o}\} \\
&\delta(q_0, \mathtt{a}) = q_0 q_0 \qquad \delta(q_0, \mathtt{b}) = q_1 \qquad \delta(q_1, \mathtt{b}) = q_1 \\
&\delta(q_0, \mathtt{c}) = \epsilon \qquad \delta(q_1, \mathtt{c}) = \epsilon
\end{aligned}
$$

$\mathcal{B}_0$ accepts Σ-labeled trees whose paths are labeled by elements of $a^{\omega} + a^*b^{\omega} + a^*b^*c$. The run tree of $\mathcal{B}_0$ over the tree generated by the recursion scheme in Example 2.1 is depicted as follows.



□

The complement of the language of trees accepted by a deterministic tree automaton is accepted by a *disjunctive alternating parity tree automaton* [19]. Since the model checking of recursion schemes for disjunctive alternating parity tree automata is $(n-1)$-EXPTIME, so is the model checking problem for deterministic tree automata. It is also $(n-1)$-EXPTIME hard, since the reachability problem is $(n-1)$-EXPTIME hard [19].

THEOREM 2.2. *Let $\mathcal{G}$ be a recursion scheme of order n, and $\mathcal{B}$ be a deterministic tree automaton. The problem of checking whether $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}$ is $(n-1)$-EXPTIME-complete.*

2.2 Model Checking Recursion Schemes and Verification of Functional Programs

As shown by Kobayashi [18], the verification of temporal properties of higher-order functional programs can be reduced to a model-checking problem for recursion schemes. The idea is to transform a functional program into a recursion scheme that generates all the possible event sequences of the program (where "events" can be calls of certain functions, resource accesses, etc., depending on the property to be verified). That is achieved by the CPS transformation. For example, consider the following program.

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

Here, `*` denotes a random boolean value. Suppose that we want to verify that the opened file is accessed according to $\mathtt{read}^*\mathtt{close}$. The above program can be transformed into the following recursion scheme $\mathcal{G}_1$, which generates a tree whose paths represent access

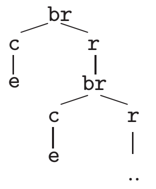


Figure 2. The tree generated by $\mathcal{G}_1$

sequences to the file.

$$\begin{array}{l} S \to G\,\mathtt{d}\,\mathtt{e} \\ G\,x\,k \to \mathtt{br}\,(\mathtt{c}\,k)\,(\mathtt{r}(G\,x\,k)) \end{array}$$

Here, the second rule corresponds to the definition of the recursive function `g`. The extra argument k represents the access sequences of the *continuation* of `g`. A non-deterministic branch is represented by the terminal `br`. The close operation has been replaced by $(\mathtt{c}\,k)$, which means that the file is closed and then it is accessed according to the continuation k. Thus, the rule for G is essentially a continuation passing style representation of the definition of `g`. The rule for S corresponds to the remaining part of the program, where `d` is a dummy terminal symbol for representing the file, and `e` expresses the program termination. The recursion scheme generates the tree shown in Figure 2. As every path from the root of the tree is labeled by an element of $((\mathtt{r}{+}\mathtt{br})^*\mathtt{c}\,\mathtt{e}){+}(\mathtt{r}{+}\mathtt{br})^\omega$, we know that the original program accesses the file in a valid manner.

Kobayashi [18] gave a systematic transformation of the resource usage verification problem [12] into a model-checking problem for recursion schemes. In general, the transformation of a functional program into a recursion scheme is achieved by (i) applying the CPS transformation and λ-lifting to get a system of recursive function definitions in CPS form $\{f_1(\widetilde{x}) = e_1, \ldots, f_n(\widetilde{x}) = e_n\}$; and then (ii) replacing $f_i(\widetilde{x}) = e_i$ with $f_i(\widetilde{x}) \to e_i'$, where e_i' is obtained from e_i by replacing each operation of interest (such as "read" and "close" above) with the corresponding tree constructor. See [18] for the details of the transformation.

EXAMPLE 2.3. The reachability problem ("Does the execution of a closed program e reach a special command `fail`?") is an instance of the resource usage verification problem; it can therefore be reduced to a model checking problem for recursion schemes. Consider, for example, the following program.

```
let g b = if b then () else fail() in
let f b = if b then g b else g (not(b)) in
  if * then f(true) else f(false)
```

It is transformed into the following recursion scheme. (Here, booleans are transformed out; $F_1\,k$ and $F_0\,k$ correspond to $f(1)$ and $f(0)$ respectively.)

$$\begin{array}{ll} S \to \mathtt{br}\,(F_1\,\mathtt{e})\,(F_0\,\mathtt{e}) & \\ F_1\,k \to G_1\,k & F_0\,k \to G_1\,k \\ G_1\,k \to k & G_0\,k \to \mathtt{fail} \end{array}$$

By the construction of the recursion scheme, the original program reaches the fail command if, and only if, the tree generated by the recursion scheme contains the symbol `fail`.

2.3 Kobayashi's Type System Equivalent to Model Checking Recursion Schemes

Kobayashi [18] has shown that the model checking problem of whether the tree generated by a recursion scheme is accepted by a trivial automaton can be reduced to a type-checking problem. The idea of the type system is to refine sorts into intersection types as sketched below.

Let $\mathcal{B}$ be a deterministic trivial automaton (Σ, Q, δ, q_0). The set of *atomic types*, ranged over by τ, is given by:

$$\tau ::= q \mid (\bigwedge\{\tau_1, \ldots, \tau_n\}) \to \tau$$

where q ranges over Q. We often write $\bigwedge_{i=1}^n \tau_i$ for $\bigwedge\{\tau_1, \ldots, \tau_n\}$, and write $\top$ for $\bigwedge \emptyset$. We give a higher precedence to $\bigwedge$ than to $\to$, and just write $\bigwedge_{i=1}^n \tau_i \to \tau$ for $(\bigwedge\{\tau_1, \ldots, \tau_n\}) \to \tau$. The type q is a refinement of the sort $\mathtt{o}$, and describes trees accepted by the automaton from state q. The type $q_1 \wedge q_2 \to q$ is a refinement of the sort $\mathtt{o} \to \mathtt{o}$; it describes functions that take, as input, a tree accepted from both states q_1 and q_2, and return a tree accepted from state q.

A type judgment for terms (where a non-terminal is treated as a variable) is of the form $\Gamma \vdash t : \tau$, where Γ, called a *type environment*, is a finite set of bindings of the form $x : \tau$. Γ may contain more than one bindings for each variable.

The typing rules are given by:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\delta(q, a) = q_1 \cdots q_n}{\Gamma \vdash a : q_1 \to \cdots \to q_n \to q}$$

$$\frac{\Gamma \vdash t_0 : \bigwedge_{i=1}^n \tau_i \to \tau \qquad \Gamma \vdash t_1 : \tau_i \text{ (for each } i = 1, \ldots, n)}{\Gamma \vdash t_0 t_1 : \tau}$$

$$\frac{\Gamma, x : \tau_1, \ldots, x : \tau_n \vdash t : \tau \qquad x \text{ not occur in } \Gamma}{\Gamma \vdash \lambda x.t : \bigwedge_{i=1}^n \tau_i \to \tau}$$

Let $\mathcal{G}$ be a recursion scheme $(\Sigma, \mathcal{N}, \mathcal{R}, S)$. We write $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ just if $\Gamma \vdash \mathcal{R}(F) : \tau$ holds for every $F : \tau \in \Gamma$. A recursion scheme $\mathcal{G}$ is well-typed, written $\vdash_{\mathcal{B}} \mathcal{G}$, just if there exists Γ such that $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ and $S : q_0 \in \Gamma$. We say "$\mathcal{G}$ is well-typed under Γ" for such Γ.

As shown in [18], the type system is sound and complete.

THEOREM 2.3 ([18]). *Let $\mathcal{B}$ be a deterministic trivial automaton, and $\mathcal{G}$ be a recursion scheme having the same terminal symbols as $\mathcal{B}$. Then, $\vdash_{\mathcal{B}} \mathcal{G}$ if, and only if, $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}$.*

Kobayashi [18] gave the following straightforward type checking (i.e. model checking) algorithm.

THEOREM 2.4 ([18]). *Let $\mathcal{F}$ be the function on type environments defined by:*

$$\mathcal{F}(\Gamma) = \{F : \tau \in \Gamma \mid \Gamma \vdash \mathcal{R}(F) : \tau\}.$$

Let Γ_{max} be the type environment:

$$\{F : \tau \mid F \in \mathit{dom}(\mathcal{N}), \textit{ and } \tau \textit{ has sort } \mathcal{N}(F)\}.$$

If $\mathcal{F}^n(\Gamma_{max}) = \mathcal{F}^{n+1}(\Gamma_{max})$ for some n,[4] *then $\vdash_{\mathcal{B}} \mathcal{G}$ if and only if $S : q_0 \in \mathcal{F}^n(\Gamma_{max})$.*

If the sizes of sorts and $|Q|$ are bounded above by a constant, then the number of bindings on each symbol in Γ_{max} is also bounded by a constant, so that the algorithm is quadratic in the size of the recursion scheme. One can further optimize the algorithm to obtain a linear time algorithm by using a standard technique for fixed-point computation [26]. In general, however, the size of Γ_{max} is $O(|\mathcal{G}|\mathbf{exp}_n((|Q|A)^{1+\epsilon}))$, where n is the order of the recursion scheme, $|\mathcal{G}|$ is the size of the recursion scheme, and A is the largest

[4] Note that such n always exists due to the finiteness of Γ_{max} and the monotonicity of $\mathcal{F}$.

```
Init:
  $\mathcal{C}$ := the initial configuration graph;
  goto Step 1;

Step 1:
  count := 0;
  while(count<MAX and an open node exists) do
  { $N$ := an open node;
    if $\mathcal{C}$ can be expanded wrt $N$ then {
      $\mathcal{C}$ := expand($\mathcal{C}, N$);
      count := count+1}
    else {
      error_path := the path from the root to $N$;
      raise PROPERTY_VIOLATED(error_path)}
  };
  goto Step 2;

Step 2:
  $\Gamma$ := $ElimTE(\Gamma_{\mathcal{C}})$;
  goto Step 3;

Step 3:
  while($\Gamma \neq \mathcal{F}(\Gamma)$) do $\Gamma := \mathcal{F}(\Gamma)$;
  if $S : q_0 \in \Gamma$ then
    raise PROPERTY_SATISFIED($\Gamma$)
  else
    goto Step 1;
```

Figure 3. Model Checking Algorithm

arity of terminal and non-terminal symbols; $\mathbf{exp}_n(x)$ is defined by $\mathbf{exp}_0(x) = x$ and $\mathbf{exp}_{i+1}(x) = 2^{\mathbf{exp}_i(x)}$. Thus, the naive algorithm above is impractical: it suffers from the n-EXPTIME bottleneck in not only the worst but also the *best* case.

REMARK 2.1. For the class of deterministic trivial automata, we can actually optimize Kobayashi's algorithm so that it runs in $(n-1)$-EXPTIME for $n \geq 2$. Still, the algorithm *always* (not just in the worst case) suffers from $(n-1)$-EXPTIME bottleneck.

3. Model-Checking Algorithm

This section describes the new model checking algorithm sketched in Figure 1. Pseudo code for the overall algorithm is shown in Figure 3. We explain Steps 3, 1, and 2 in this order.

3.1 Step 3

Step 3 takes a type environment Γ as an input, and checks whether there exists a subset Γ' of Γ such that $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma'$ and $S : q_0 \in \Gamma'$. In the pseudo code shown in Figure 3, $\mathcal{F}$ is the function on type environments defined in Theorem 2.4.

The following theorem (which follows from the standard fixed-point theorem) guarantees the correctness of the algorithm.

THEOREM 3.1. *Let $\mathcal{F}$ be the function on type environments as defined in Theorem 2.4. Then, $\mathcal{F}^0(\Gamma)(=\Gamma), \mathcal{F}^1(\Gamma), \mathcal{F}^2(\Gamma), \ldots$ is a decreasing sequence, i.e.*

$$\mathcal{F}^0(\Gamma) \supseteq \mathcal{F}^1(\Gamma) \supseteq \mathcal{F}^2(\Gamma) \supseteq \cdots ;$$

and there exists n such that $\mathcal{F}^n(\Gamma) = \mathcal{F}^{n+1}(\Gamma)$. Furthermore, $\Gamma' = \mathcal{F}^n(\Gamma)$ is the largest set such that $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma'$ and $\Gamma' \subseteq \Gamma$.

3.2 Step 1

The role of Steps 1 and 2 is to find a good candidate for the type environment, which should be passed to Step 3 described above. An obvious candidate is Γ_{max} introduced in Section 2.3, but that is not a good choice, as the size of Γ_{max} is n-exponential, which grows up very quickly. For example, suppose that the number of automaton states is 2. Then the number of intersection types for each sort grows as follows.

sorts	the number of intersection types
$\mathtt{o} \rightarrow \mathtt{o}$	$2^2 \times 2 = 8$
$(\mathtt{o} \rightarrow \mathtt{o}) \rightarrow \mathtt{o}$	$2^8 \times 2 = 512$
$((\mathtt{o} \rightarrow \mathtt{o}) \rightarrow \mathtt{o}) \rightarrow \mathtt{o}$	$2^{512} \times 2 = 2^{513} \approx 10^{154}$

Thus, Kobayashi's previous type-checking (i.e. model checking) algorithm [18] would be impractical even for order-3 recursion schemes.

As mentioned in Section 1, the key observation behind our new algorithm is that, in realistic programs, the usage patterns of each function are limited, so that the size of the type environment that should be passed to Step 3 can be much smaller than the size of Γ_{max}. To guess such a good type environment, in Step 1, we reduce the given recursion scheme a finite number of steps and construct a configuration graph (representing the traces of the reduction). We then extract type information from the configuration graph in Step 2.

A *configuration graph* is a labeled directed graph, constructed by applying the expansion operations defined below. The initial configuration graph is a graph consisting of just a single node (called the *root node*), labeled by $\langle S, q_0, \mathtt{open}\rangle$. Let $\mathcal{C}$ be a configuration graph, and N is a node of $\mathcal{C}$, labeled by $\langle t, q, \mathtt{open}\rangle$. Then, an *expansion* of $\mathcal{C}$ wrt N is the graph $\mathcal{C}'$ obtained from $\mathcal{C}$ by replacing the flag of N with $\mathtt{closed}$, and adding nodes and (directed) edges as follows.

(I) Case $t = a\,t_1 \,\cdots\, t_m$ and $\delta(q, a) = q_1 \cdots q_m$:
If a node labeled with $\langle t_i, q_i, f_i\rangle$ (for some f_i) does not exist, add a new node $\langle t_i, q_i, \mathtt{open}\rangle$. Add directed edges from N to the nodes labeled by $\langle t_1, q_1, f_1\rangle, \ldots, \langle t_m, q_m, f_m\rangle$, and label each edge from N to the node $\langle t_i, q_i, f_i\rangle$ with i.

(II) Case $t = F\,t_1 \,\cdots\, t_m$ and $\mathcal{R}(F) = \lambda x_1. \cdots \lambda x_m.t$:
If a node labeled with $\langle [t_1/x_1, \ldots, t_m/x_m]t, q, f\rangle$ does not exist, add a new node $\langle [t_1/x_1, \ldots, t_m/x_m]t, q, \mathtt{open}\rangle$. Add a directed edge from N to the node labeled by $\langle [t_1/x_1, \ldots, t_m/x_m]t, q, f\rangle$, and label the edge with 0.

The configuration graphs of a recursion scheme are those obtained from the initial configuration graph by applying (possibly an infinite number of) expansion operations. A configuration graph is *finitely-expanded* if it is obtained by a finite number of expansions. A configuration graph is *closed* if it contains no open node (i.e. node which is labeled by $\langle t, q, \mathtt{open}\rangle$). Note that for a recursion scheme, a closed configuration graph is uniquely determined (up to the graph isomorphism), and it may be an infinite graph.

We often write $[t, q]$ for $\langle t, q, \mathtt{closed}\rangle$ and (t, q) for $\langle t, q, \mathtt{open}\rangle$. Figure 4 shows a configuration graph for the recursion scheme $\mathcal{G}_0$ in Example 2.1. The edge label 0 is omitted in the figure. The only open node is $(F(\mathtt{b}(\mathtt{b}\,c)), q_0)$.

Figure 3 shows the pseudo code for Step 1. Several nodes are expanded and the resulting configuration is passed to Step 2. A graph cannot be expanded wrt N if N is labeled by $\langle a\,t_1 \,\cdots\, t_m, q, \mathtt{open}\rangle$ but $\delta(q, a)$ is undefined. In that case, the property is violated, so that the path from the root to the node is output as an *error path*. The selection of the node N on line 3 of Step 1 must be fair, in the sense that every open node (i.e. a node labeled with open) must be eventually selected. (The fairness can be easily ensured, for example, by maintaining a FIFO queue of open nodes.)

EXAMPLE 3.1. *Consider the recursion scheme $\mathcal{G}_1$ given by the following rewriting rules:*

$$S \rightarrow F(F\,\mathtt{c}) \qquad F\,x \rightarrow a\,x\,(b(F\,x)),$$

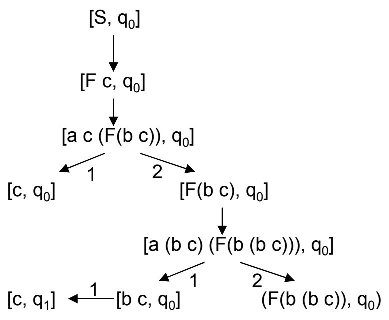


Figure 4. A Configuration Graph

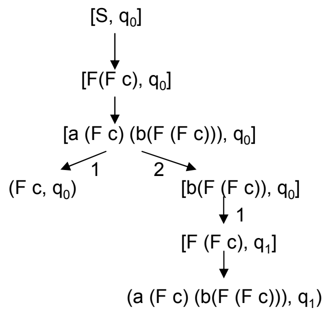


Figure 5. A Configuration Graph Containing an Error Node

and the automaton $\mathcal{B}_0$ in Example 2.2. The configuration graph obtained by several expansions is shown in Figure 5. The node at the bottom of the figure cannot be expanded, as $\delta(q_1, \mathtt{a})$ is undefined. The error path is 0:0:2:1:0 *(where ":" denotes the string concatenation). In fact, the path* $2:1$ *(obtained by ignoring* 0*) of the tree generated by the recursion scheme is labeled by* aba*, which violates the property expressed by $\mathcal{B}_0$, that* a *should not occur after* b*.*

3.3 Step 2

We now describe the most important part of the algorithm: Step 2 for extracting type information from a configuration graph.

As mentioned in Section 1, the algorithm for extracting type information described below has been inspired from the proof of the completeness of Kobayashi and Ong's type system for the modal μ-calculus model checking of recursion schemes [20]. We call a term t' a *prefix* of t if t is of the form $t'\,\widetilde{s}$, where $\widetilde{s}$ is a possibly empty sequence of terms. For each node N labeled with $\langle t, q, f\rangle$, we assign a type $\tau_{t',N}$ to each prefix t' of t. The type $\tau_{t,N}$ is defined by induction on the sort of t as follows.

(I) Case t has sort $\mathtt{o}$:
In this case, N must have a label of the form $\langle t, q, f\rangle$. Define $\tau_{t,N} := q$.

(II) Case t has sort $\kappa_1 \to \kappa_2$:
In this case, N must have a label of the form $\langle t\, s_0\, \widetilde{s}, q, f\rangle$, where s_0 and $t\,s_0$ have sorts κ_1 and κ_2 respectively. Let $\{N_1, \ldots, N_m\}$ be the set of nodes that are reachable from N, and have labels of the form $\langle s_0\, \widetilde{u}, q', f\rangle$ (where s_0 must originate from that of the node N; thus we assume implicitly that a configuration graph has a link to show the origin of each term). If s_0 occurs in an open node reachable from N (including N itself), then let $S := \{\alpha, \tau_{s_0,N_1}, \ldots, \tau_{s_0,N_m}\}$ where α is a fresh type variable (which indicates that the type can be further refined by additional expansion of the configuration graph). Otherwise, let $S := \{\tau_{s_0,N_1}, \ldots, \tau_{s_0,N_m}\}$. Finally, define $\tau_{t,N} := (\bigwedge S) \to \tau_{t\, s_0,N}$. (Note that the sorts of ts_0 and s_0 are κ_2 and κ_1 respectively, so that $\tau_{t\, s_0,N}$ and τ_{s_0,N_i} are determined by the induction.)

Now, given a configuration graph $\mathcal{C}$, define the type environment $\Gamma_{\mathcal{C}}$ by:

$$\Gamma_{\mathcal{C}} := \{F : \tau_{F,N} \mid N \text{ has a label of the form } \langle F t_1 \cdots t_m, q, f\rangle\}.$$

EXAMPLE 3.2. *Consider the configuration graph $\mathcal{C}$ of Figure 4. Let $N_0, N_1, N_2, N_3, N_4,$ and N_5 be the nodes labeled by $[S, q_0]$, $[F\,\mathtt{c}, q_0]$, $[\mathtt{c}, q_0]$, $[F\,(\mathtt{b}\,\mathtt{c}), q_0]$, $[\mathtt{b}\,\mathtt{c}, q_0]$, and $[\mathtt{c}, q_1]$ respectively. Then,*

$$\begin{aligned}
&\tau_{S,N_0} = q_0\\
&\tau_{F,N_1} = \bigwedge\{\tau_{\mathtt{c},N_2}, \tau_{\mathtt{c},N_5}, \alpha\} \to \tau_{F\,\mathtt{c},N_1} = (q_0 \wedge q_1 \wedge \alpha) \to q_0\\
&\tau_{F,N_3} = \bigwedge\{\tau_{\mathtt{b}\,\mathtt{c},N_4}, \alpha'\} \to \tau_{F\,(\mathtt{b}\,\mathtt{c}),N_3} = (q_0 \wedge \alpha') \to q_0
\end{aligned}$$

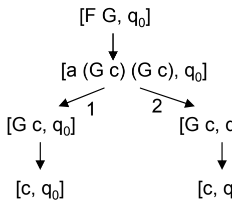
Here, the type variables α and α' denote the types of c *and* b c *respectively, in the open node labeled by $(F\,(\mathtt{b}\,(\mathtt{b}\,\mathtt{c})), q_0)$.*

$\Gamma_{\mathcal{C}}$ is given by:

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

□

EXAMPLE 3.3. *As an example of higher-order case, consider the following fragment of a configuration graph:*



Let N_0, N_1, N_2, N_3, and N_4 be the nodes labeled by $[F\,G, q_0]$, $[G\,\mathtt{c}, q_0]$, $[\mathtt{c}, q_0]$, $[G\,\mathtt{c}, q_1]$, and $[\mathtt{c}, q_1]$. τ_{F,N_0} is computed as follows.

$$\begin{aligned}
\tau_{F,N_0} &= \bigwedge\{\tau_{G,N_1}, \tau_{G,N_3}\} \to \tau_{F\,G,N_0}\\
&= \bigwedge\{\tau_{\mathtt{c},N_2} \to \tau_{G\,\mathtt{c},N_1}, \tau_{\mathtt{c},N_4} \to \tau_{G\,\mathtt{c},N_3}\} \to q_0\\
&= \bigwedge\{q_0 \to q_0, q_1 \to q_1\} \to q_0
\end{aligned}$$

□

The following is the key theorem underlying the algorithm of Step 2, which follows from the proof of the completeness of Kobayashi and Ong's type system [20].[5]

THEOREM 3.2. *Suppose that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}$. Then, there exists a closed (possibly infinite) configuration graph $\mathcal{C}$ of $\mathcal{G}$ over $\mathcal{B}$. Furthermore, $\mathcal{G}$ is well-typed under $\Gamma_{\mathcal{C}}$.*

The above theorem cannot be directly used in the algorithm, since, in general, infinitely many expansions are necessary to get a closed configuration graph $\mathcal{C}$. By the definition of $\Gamma_{\mathcal{C}}$, however, a finitely expanded graph provides an approximation of $\Gamma_{\mathcal{C}}$.

LEMMA 3.3. *Let $\mathcal{C}'$ be a finitely expanded configuration graph of $\mathcal{G}$ over $\mathcal{B}$, and $\mathcal{C}$ be the closed configuration graph. Then, there is a substitution θ for type variables such that $\theta\Gamma_{\mathcal{C}'} = \Gamma_{\mathcal{C}}$.*

As a corollary of Theorem 3.2 and Lemma 3.3, we obtain:

[5] More precisely, Kobayashi and Ong [20] considered a run tree corresponding to the closed configuration graph, gave the definition of $\tau_{F,N}$, and proved a theorem corresponding to Theorem 3.2. They did not consider how to extract type information from a finitely-expanded configuration graph.

COROLLARY 3.4. *Suppose that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}$. Let $\mathcal{C}$ be a configuration graph of $\mathcal{G}$ over $\mathcal{B}$. Then, there is a substitution θ for type variables such that $\mathcal{G}$ is well-typed under $\theta\Gamma_{\mathcal{C}}$, i.e. $\vdash_{\mathcal{B}} \mathcal{G} : \theta\Gamma_{\mathcal{C}}$ and $S : q_0 \in \theta\Gamma_{\mathcal{C}}$ holds.*

In Step 2 (see Figure 3), therefore, $\Gamma_{\mathcal{C}}$ is first computed from the current (finitely expanded) configuration graph $\mathcal{C}$. The type variables are then removed from $\Gamma_{\mathcal{C}}$ by the operation *ElimTE* described below.

We first define the function *Elim*, which takes an atomic type as input, and returns a set of closed atomic types.

$$\begin{array}{l} \mathit{Elim}(q) = \{q\} \\ \mathit{Elim}(\alpha) = \emptyset \\ \mathit{Elim}(\bigwedge_{i=1}^{m} \tau_i \rightarrow \tau) = \\ \qquad \{\bigwedge_{i=1}^{m} \tau'_i \rightarrow \tau' \mid \tau'_i \in \mathit{Elim}'(\tau_i), \tau \in \mathit{Elim}(\tau)\} \\ \mathit{Elim}'(\tau) = \begin{cases} \mathit{Elim}(\tau) \cup \{\top\} & \text{if } \tau \text{ contains a type variable} \\ \mathit{Elim}(\tau) & \text{otherwise} \end{cases} \end{array}$$

The auxiliary function Elim' may return a set consisting of closed atomic types and $\top$; we treat $\top$ as the unit on $\wedge$, i.e. we identify $\bigwedge\{\tau_1, \ldots, \tau_n, \top\}$ with $\bigwedge\{\tau_1, \ldots, \tau_n\}$.

In the definition of $\mathit{Elim}(\bigwedge_{i=1}^{m} \tau_i \rightarrow \tau)$, if an argument type τ_i contains a type variable, *Elim* may choose $\top$ from $\mathit{Elim}'(\tau_i)$ (intuitively, because τ_i may express incomplete information that should be ignored). For example, we have:

$$\begin{array}{l} \mathit{Elim}((q_0 \wedge q_1 \rightarrow q_2) \wedge (q_0 \wedge \alpha \rightarrow q_2) \rightarrow q) \\ = \{\tau_1 \wedge \tau_2 \rightarrow q \mid \\ \qquad \tau_1 \in \mathit{Elim}'(q_0 \wedge q_1 \rightarrow q_2), \tau_2 \in \mathit{Elim}'(q_0 \wedge \alpha \rightarrow q_2)\} \\ = \{((q_0 \wedge q_1 \rightarrow q_2) \wedge \tau_2) \rightarrow q \mid \tau_2 \in \{q_0 \rightarrow q_2, \top\}\} \\ = \{(q_0 \wedge q_1 \rightarrow q_2) \rightarrow q, (q_0 \wedge q_1 \rightarrow q_2) \wedge (q_0 \rightarrow q_2) \rightarrow q\} \end{array}$$

ElimTE is a pointwise extension of the operation *Elim*, defined by:

$$\mathit{ElimTE}(\Gamma) = \bigcup_{F:\tau \in \Gamma} \{F : \tau' \mid \tau' \in \mathit{Elim}(\tau)\}$$

REMARK 3.1. The completeness is lost if we simply replace all the type variables in Γ with $\top$, instead of applying the operation *ElimTE* above. For example, suppose $\Gamma = \{F : (q_0 \wedge q_1 \rightarrow q_2) \wedge (q_0 \wedge \alpha \rightarrow q_2) \rightarrow q\}$. If we replace α with $\top$, we obtain the type environment $\{F : (q_0 \wedge q_1 \rightarrow q_2) \wedge (q_0 \rightarrow q_2) \rightarrow q\}$. However, the actual type of F in the recursion scheme may be $(q_0 \wedge q_1 \rightarrow q_2) \rightarrow q$.

EXAMPLE 3.4. *Recall the type environment $\Gamma_{\mathcal{C}}$ in Example 3.2. ElimTE($\Gamma_{\mathcal{C}}$) is:*

$$\{S : q_0, F : q_0 \wedge q_1 \rightarrow q_2, F : q_0 \rightarrow q_2\}.$$

□

Optimizations The operation *Elim* above on function types may cause a combinatorial explosion of the number of atomic types. The following optimizations reduce the number of atomic types without losing the completeness of the algorithm.

(I) Use the canonical representation of function types. Here, a function type $\bigwedge_{i=1}^{n} \tau_i \rightarrow \tau$ is canonical if for each i, there is no $j \neq i$ such that $\tau_j \leq \tau_i$ (where $\leq$ is the standard subtype relation). (Accordingly, we need to extend the type system used in Step 3 with subtyping.)

(II) In the definition of $\mathit{Elim}(\bigwedge_{i=1}^{n} \tau_i \rightarrow \tau)$, choose $\top$ as τ'_i only if τ_i is subsumed by τ_j for some j, i.e. if there is a substitution θ such that $\theta\tau_i = \tau'_j$ for some j.
Both optimizations are applied in the current implementation of the model checker described in Section 4.

EXAMPLE 3.5. *Let τ be:*

$$(q_0 \wedge q_1 \rightarrow q_2) \wedge (\alpha_0 \wedge q_1 \rightarrow q_2) \wedge (q_0 \wedge \alpha_1 \rightarrow q_2) \wedge (\alpha_2 \rightarrow q_2) \rightarrow q.$$

$\mathit{Elim}(\tau)$ generates the following set of types:

$$\{\bigwedge(S \cup \{q_0 \wedge q_1 \rightarrow q_2\}) \rightarrow q \mid S \subseteq \{q_1 \rightarrow q_2, q_0 \rightarrow q_2, \top \rightarrow q_2\}\}.$$

After the optimizations, $\mathit{Elim}(\tau)$ generates:

$$\{\tau \rightarrow q \mid \tau \in \{q_0 \wedge q_1 \rightarrow q_2, q_1 \rightarrow q_2, q_0 \rightarrow q_2, \top \rightarrow q_2\}\}.$$

3.4 Correctness of the Algorithm

We now prove the soundness and completeness of the algorithm.

THEOREM 3.5. *Given a recursion scheme $\mathcal{G}$, and a deterministic trivial automaton $\mathcal{B}$, the algorithm eventually terminates. Furthermore, if the algorithm outputs a type environment Γ, then $\mathcal{G}$ is well-typed under Γ (hence $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}$). If the algorithm reports an error path, then $[\![\mathcal{G}]\!]$ is not accepted by $\mathcal{B}$.*

Proof The soundness of the output follows immediately from Theorem 3.1 and the definition of Step 1. By the construction of the configuration graph in Step 1 and the fairness assumption on the selection of nodes, if $[\![\mathcal{G}]\!]$ is not accepted by $\mathcal{B}$, then an error path is eventually found. Thus, it remains to show that the algorithm eventually terminates when $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}$.

Let $\mathcal{C}$ be the closed configuration graph. By Corollary 3.2, it suffices to show that there exists a finitely-expanded configuration graph $\mathcal{C}'$ such that $\mathit{ElimTE}(\Gamma_{\mathcal{C}'}) \supseteq \Gamma_{\mathcal{C}}$.

Let π be a sequence over $\{0, 1, \ldots, m\}$ where m is the largest arity of terminals. We write $\mathcal{C}(\pi)$ for the node N such that a path from the root to N is labeled by π. Let $\mathcal{C}$ be the closed configuration graph and $\mathcal{C}'$ be a finitely-expanded graph. Suppose that the node $\mathcal{C}(\pi)$ is labeled by $\langle t\,\widetilde{s}, q, \texttt{closed}\rangle$. We define the relation $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$ by induction on the structure of the sort of t, as follows.

(i) If t has sort $\mathtt{o}$, and $\tau_{t,\mathcal{C}'(\pi)} = q$, then $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$.

(ii) Suppose t has sort $\kappa_1 \rightarrow \kappa_2$. Then, $\widetilde{s}$ must be of the form $s_0\widetilde{s}'$, and $\tau_{t,\mathcal{C}(\pi)}$ is of the form $\bigwedge_{i=1}^{n} \tau_i \rightarrow \tau$. $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$ holds if the following conditions are satisfied:
(a) $\mathcal{C}' \preceq_{\pi,t\,s_0} \mathcal{C}$;
(b) for each τ_i, there exists a path π_i such that $\tau_{s_0,\mathcal{C}(\pi\pi_i)} = \tau_i$ and $\mathcal{C}' \preceq_{\pi\pi_i,s_0} \mathcal{C}$.
Intuitively, $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$ means that $\mathcal{C}'$ provides complete type information about how the term t of node $\mathcal{C}(\pi)$ is used in $\mathcal{C}$.

Suppose $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$. The following properties can be proved by induction on the structure of the sort of t: See Appendix A, Lemma A.1.

(I) If $\mathcal{C}''$ is obtained from expansions of $\mathcal{C}'$, then $\mathcal{C}'' \preceq_{\pi,t} \mathcal{C}$.

(II) $\tau_{t,\mathcal{C}(\pi)} \in \mathit{Elim}(\tau_{t,\mathcal{C}'(\pi)})$.
We can also prove, by induction on the sort of t, that for any node $\mathcal{C}(\pi)$ labeled with $\langle t\,\widetilde{s}, q, \texttt{closed}\rangle$, there exists a finitely-expanded graph $\mathcal{C}'$ such that $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$ (see Appendix A, Lemma A.2).

Now, for each $F_i : \tau_{i,j} \in \Gamma_{\mathcal{C}}$, pick a node $N_{i,j} = \mathcal{C}(\pi_{i,j})$ such that $\tau_{F_i,N_{i,j}} = \tau_{i,j}$. Let $\mathcal{C}_{i,j}$ be a finitely-expanded graph such that $\mathcal{C}_{i,j} \preceq_{\pi_{i,j},F_i} \mathcal{C}$, and let $\mathcal{C}'$ be the union of all such $\mathcal{C}_{i,j}$'s. (By the "union" of graphs, we mean the graph obtained by merging the corresponding nodes into one node, where its flag is set to `closed` if one of the nodes is `closed`. In other words, the union of configuration graphs $\mathcal{C}_1, \ldots, \mathcal{C}_m$ is obtained from the initial graph by expanding all the closed nodes of $\mathcal{C}_i$'s.) By the property (I) above, $\mathcal{C}' \preceq_{\pi_{i,j},F_i} \mathcal{C}$ for every i, j. Thus, by the property (II), we have $\Gamma_{\mathcal{C}} \subseteq \mathit{ElimTE}(\mathcal{C}')$ as required. □

4. Preliminary Experiments

We have implemented a model checker TRECS (Types for RECursion Scheme) for recursion schemes. The implementation

Programs	O	R	S	Q	result	E	time
`Example 2.1`	1	2	8	2	YES	53	1
`Example 2.3`	1	6	13	1	YES	40	1
`Example 3.1`	1	2	8	1	NO	9	1
`File`	1	2	8	2	YES	46	1
`Flow`	3	7	16	1	YES	8	1
`Exception`	1	5	18	1	YES	7	1
`Twofiles`	4	11	47	4	YES	153	2
`FileWrong`	4	11	45	4	NO	66	1
`TwofilesExn`	4	12	56	5	YES	189	2
`FileOcamlc`	4	23	110	4	YES	254	5
`Lock1`	4	12	38	3	YES	34	1
`Lock2`	4	11	45	4	YES	255	5
`Order5`	5	11	52	5	YES	165	2
`Order5-2`	5	9	36	4	YES	198	3

Table 2. Experimental results (time is in milliseconds).

can be tested at `http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/`.[6]

The implementation is mostly based on the algorithm described in Section 3, except the following points:

- The current implementation uses trees for representing configuration graphs; thus, the same node may be expanded multiple times, which causes a performance bottleneck. This is just for the sake of simplicity and will be improved in a future version.
- To select a node to be expanded in Step 1, we use the depth-first search, except that the expansions of nodes for recursive calls are delayed. The number of iterations in Step 1 (the value of `MAX` in Figure 3) is set to 100. (There is no particular reason for the choice of 100; the value of `MAX` only affects the performance.)
- The two optimizations discussed at the end of Section 3.3 are applied.
- To ease the extraction of type information in Step 2, the implementation keeps pointers to express the origin of each term. For example, in Figure 4, we keep a pointer from c of the node $[\mathtt{a}\,\mathtt{c}\,(F(\mathtt{b}\,\mathtt{c})), q_0]$ to that of the node $[F\,\mathtt{c}, q_0]$.
- In Step 3, Rehof and Mogensen's algorithm is used for the fixed-point computation.
- We keep the type environment obtained in Step 3, and use it to eliminate open nodes that are typable under the type environment. For example, consider an open node labeled by $(F\,G, q_0)$. If the type environment obtained in the previous cycle contains the bindings $F : q_0 \rightarrow q_0, G : q_0$, we remove the open node, recording that F and G have been used as terms of types $q_0 \rightarrow q_0$ and q_0 respectively.

We have tested the model checker for a number of small but tricky examples. Table 2 shows the result. The columns O, R, S, and Q show the order of the recursion scheme, the number of rewriting rules, the size of rewriting rules (which are measured by the number of occurrences of symbols in the righthand side of the rewriting rules) and the number of automaton states respectively. The column "result" shows whether the property is satisfied (YES) or not (NO). (Note that, in addition to yes/no answers, the model checker outputs a type environment if the property is satisfied, and reports an error trace otherwise.) The column "E" shows the number of expansions of the configuration graph. The column "time" shows the running time, measured in milliseconds. The experiment was conducted on a machine with Intel(R) Xeon(R) CPU with 3Ghz and 2GB memory.

[6] Only small examples can be tested through the web interface. The source code will be distributed later.

The recursion schemes used in the experiments are available from `http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/`. The first four programs (or, recursion schemes) were taken from earlier sections of this paper. For the first one, the specification is the automaton of Example 2.2. The fourth example "File" is the recursion scheme given in Section 2.2, obtained from the program accessing a file.

The program `Flow` encodes the following flow analysis problem, taken from [22].

```
let id x = x in let unused = id lam in
  id lam'
```

Here, the flow question is whether "lam" flows to the result of the program. We can encode this into the problem of whether the terminal symbol `flow` occurs in the tree generated by the following recursion scheme.

```
S -> C1 Id.
Id x k -> k x.
C1 id -> id Lam (C2 id).
C2 id unused -> id LamPrime C3.
C3 x -> x end.
Lam x -> flow x.
LamPrime x -> x.
```

The first four rules are obtained by applying the CPS transformation to the original program. A trick is in the last three rules. The value of the program is passed to `C3`. If the value is `Lam`, then `Lam` is invoked and the terminal symbol `flow` occurs in the value tree; otherwise, `flow` does not occur. Thus, `Lam` flows to the value of the original program if, and only if, `flow` occurs in the tree generated by the recursion scheme. In general, any instance of the flow analysis problem ("Given a program and its subterms v and e, does the value of v flow to the value of e?") can be encoded into a resource usage verification problem (by replacing each value with a pair consisting of the value and a resource to keep track of its use[7]), and then to a model checking problem of a recursion scheme.

The program `Exception` encodes the following exception analysis problem (of checking whether an uncaught exception may be raised), taken from [21]:

```
let failwith msg = raise(Failure msg)
let f x = if ... then ... else failwith ("f")
let g x = try f x with Failure "f" -> 0
let main() = g()
```

By representing exception handlers as alternative continuations as in [7], we get the following recursion scheme.

```
Failwith0 failure0 failure1 k -> failure0
Failwith1 failure0 failure1 k -> failure1
F failure0 failure1 k ->
     br k (Failwith0 failure0 failure1 k)
G failure0 failure1 k -> F k failure1 k
S -> G uncaught uncaught end
```

Here, the string `"f"` has been replaced by a boolean value, and then the boolean elimination [18] has been applied. The original program raises an uncaught exception if and only if the value tree of the recursion scheme contains the terminal `uncaught`.

The 7th–12th programs were obtained from resource usage verification problems (of checking whether files and locks are used correctly) by manually applying Kobayashi's transformation [18]. `Twofiles`, taken from [18], is a program copying a read-only file to a write-only file. The corresponding OCaml-like program is:

[7] In the translation of the above example, the value part is omitted since it is not used in the original program.

```
 let rec f x y = if * then close x; close y
                 else read x; write y; f x y in
 let x = open_in "foo" in let y = open_out "bar" in
 f x y
```

FileWrong is the program obtained from Twofiles by removing the close operation on the write-only file.

TwofilesExn is based on the following program, which copies a read-only file to a write-only file, detecting the end of a file by an exception:

```
let rec f(x,y) = read(x);write(y);f(x,y) in
let x = open_in "foo" in
let y = open_out "bar" in
  try f(x,y) with end_of_file -> close(x); close(y)
```

In the recursion scheme, the exception handler is passed to f as an additional continuation argument. The rules for f and read are as follows.

```
F x y ex k -> Read x ex (Write y ex (F x y ex k))
Read x ex k -> ReadWithoutExn x (br ex k)
```

Here, ex is bound to the continuation expressing an exception handler. In the rule for Read, the part "br ex k" captures the fact that Read may or may not raise an exception.

The program FileOcamlc is based on a part of the source code of Objective Caml compiler 3.11.0 [1], bytecomp/symtable.ml, which is the most interesting and complex use of input files we found in the compiler source code. The following is a simplified version of the code:

```
let rec readloop x =
  if * then () else readloop x; read x in
let read_sect() =
  let fp = open "foo" in
  {readc = fun x -> readloop fp;
   closec = fun x -> close fp} in
let rec loop s =
 if * then s.closec() else s.readc(); loop s
in
  let s = read_sect() in  loop s
```

The function read_sect opens a file and returns closures to access the file. This kind of program was extremely hard to analyze for other (incomplete) methods for the resource usage verification [12, 13]. The original source code consists of about 60 lines of code. We obtained the recursion scheme from it by manually slicing irrelevant parts and applying the CPS transformation. (Thus, the verified recursion scheme is more complex than the simplified code above.)

The program Lock1, taken from [18], is based on the following code:

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

The rules for encoding the function f above are:

```
   F0 x k -> k.
   F1 x k -> Lock x k.
```

Here, the boolean parameter b has been transformed out (without losing information) as described in [18]; F0 and F1 simulate the behaviors of f false and f true respectively.

The program Lock2 is based on the following program.

```
 let l1 = newlock() in
 let rel1 x = unlock(l1) in
 let acq1 x = lock(l1) in
 let rec f g =
    if * then g()
    else
     let l2 = newlock() in
     let rel2 x = unlock(l2) in
     let acq2 x = lock(l2) in
       (acq2(); f rel2; g())
 in
    (acq1(); f rel1)
```

Analyzing the above program is very tricky, as (i) infinitely many locks are created, and (ii) locks are stored in closures and accessed through them.

In order to test the effectiveness of the current implementation, we have also tested verification of some order-5 recursion schemes. (Recall that the most significant parameter that determines the complexity of model-checking recursion schemes is their orders, rather than the size of the recursion schemes.) The program Order5 is based on the following program.

```
 let rec loop use finish x =
      if * then finish x
      else use x; loop use finish x in
 let gencon gen use finish =
      let x = gen() then loop use finish x in
 let genr () = open_in * in
 let genw () = open_out * in
  gencon genr read close; gencon genw write close
```

The function gencon takes a generator and consumers of resources as an argument, creates a new resource by invoking the generator, and then uses it. In the corresponding recursion schemes, gencon has order 5, and loop (at which recursion occurs) has order 4.

The program Order5-2 is based on the following program.

```
let rec gencon gen use =
    if * then ()
    else let x = gen() in
       gencon gen use; use x in
let f x = if * then close x else read x; f x in
let genr () = open_in * in
   gencon genr f
```

In the corresponding recursion schemes, gencon (at which recursion occurs) has order 5.

Our model checker could correctly verify all the recursion schemes (or reject, in case the property is not satisfied) in less than a second. This is remarkably fast, considering that the model checking of recursion schemes is $(n-1)$-EXPTIME in general (so, 3-EXPTIME for the 7th–12th programs); Note also that all the previous model-checking algorithms [18, 20, 24] are simply unexecutable for the recursion schemes of orders 4 or 5, because of huge requirement for time and space (recall the discussion in Section 3.1). Although the examples are small, at least FileOcamlc and lock2 are very complex: we are not aware of any previous automated techniques for the resource usage verification that can correctly verify them.

The followings are further observations from the experiments.

- The node selection strategy in Step 1 can significantly affect the overall performance of the model checker. For example, for FileOcamlc, the model checking has timed out when we used the pure breadth-first search.
- Without the optimizations described in Section 3.3, the model checker ran out of the stack space for the programs "FileOcamlc" and "Lock2." That is due to the combinatorial explosion of the number of atomic types, introduced by *Elim* (recall Ex-

ample 3.5). The combinatorial explosion is disastrous for symbols of high orders and large arities. After the optimizations, however, the number of atomic types is kept small.

5. Discussion

A limitation of the algorithm described in Section 3 (especially from a theoretical point of view) is that the worst-case time complexity is not optimal. According to the result of [18], a recursion scheme can be model-checked in time linear in the size of the grammar, provided that the sizes of sorts and the trivial automaton are bounded above by a constant. The worst case time complexity of the algorithm in Section 3 is, however, at least n-exponential under the same assumption. For example, consider the following recursion scheme for generating a word.

$$\begin{array}{l} S \to F_0\, G \\ F_0\, x \to F_1(F_1\, x) \\ F_1\, x \to F_2(F_2\, x) \\ \cdots \\ F_{m-1}\, x \to F_m(F_m\, x) \\ F_m\, x \to \mathtt{a}\, x \\ G \to \mathtt{c} \end{array}$$

Since S is reduced to $\mathtt{a}^{2^m} G$, and then to $\mathtt{a}^{2^m}\,\mathtt{c}$, the algorithm needs to expand the initial configuration graph $O(2^m)$ times in Step 1 to extract type information of G. In general, we can construct an order-n recursion scheme of size m for which our algorithm requires $O(\mathbf{exp}_n(m))$ expansions of configuration graphs.

To address the problem above, we can make the following modification to Step 1. Let N be a node labeled with $\langle F\, t, q_0, \mathtt{open}\rangle$ where F is a non-terminal of order 1. If there is a path π from another node N_1 labeled with $\langle F\, s, q_0, f\rangle$ to a node N_2 labeled with $\langle s, q_1, f'\rangle$ (where s originates from that of $F\, s$), then add an edge from N to $\langle t, q_1, \mathtt{open}\rangle$. This is sound because the argument s cannot be used in the path from N_1 to N_2, so that the same computation is possible from N to produce the node $\langle t, q_1, \mathtt{open}\rangle$. For the example above, we first expand S as follows.

$$\begin{array}{l} [S, q_0] \to [F_0\, G, q_0] \to [F_1(F_1\, G), q_0] \to [F_2(F_2(F_1\, G)), q_0] \\ \to \cdots \to [F_m(F_m(F_{m-1}(\cdots F_2(F_1\, G)\cdots))), q_0] \\ \to [a(F_m(F_{m-1}(\cdots F_2(F_1\, G)\cdots))), q_0] \\ \to (F_m(F_{m-1}(\cdots F_2(F_1\, G)\cdots)), q_0) \end{array}$$

Here, we assume that $\delta(q_0, a) = q_0$. Now, since there is a path from $[F_m\, s, q_0]$ to (s, q_0) for $s = F_m(F_{m-1}(\cdots F_2(F_1\, G)\cdots))$, we can add an edge from $(F_m(F_{m-1}(\cdots F_2(F_1\, G)\cdots)), q_0)$ to $(F_{m-1}(\cdots F_2(F_1\, G)\cdots), q_0)$. By applying the same construction to $F_{m-1}, \ldots, F_1$, we get

$$\begin{array}{l} [F_{m-1}(\cdots F_2(F_1\, G)\cdots), q_0] \to [F_{m-2}(\cdots F_2(F_1\, G)\cdots), q_0] \\ \to \cdots \to [F_1\, G, q_0] \to [G, q_0] \to [\mathtt{c}, q_0] \end{array}$$

Thus, we can extract the type information on G by $O(m)$ expansions of the initial graph. A similar optimization is possible for the case of the 2nd or higher order.

The above optimization alone is, however, insufficient for obtaining an algorithm with the optimal worst-case complexity. Further optimizations are left for future work.

6. Related Work

Previous work on model-checking recursion schemes Knapik et al. [15] have shown that the modal μ-calculus model checking of order-2 safe recursion schemes is decidable (where the safety is a certain syntactic restriction). This subsumes earlier results of Rabin [25] for order-0 case (i.e. for regular trees) and Courcelle [10] for order-1 case (i.e. algebraic trees). Since then, the decidability question of recursion schemes of an arbitrary order had been a hot topic in the theoretical community [3, 16] (most notably, Knapik et al.'s result on the decidability of *safe* recursion schemes [16]), until Ong [24] finally answered the question positively. Kobayashi and Ong [20] recently gave a simpler, type-based proof of the decidability.

Despite the theoretical results on the decidability of the model checking problem above, there has been little work on its application to program verification until Kobayashi's recent work [18]. This may be due to the n-EXPTIME hardness of the model checking problem. Kobayashi [18] has shown that, for the class of trivial automata, the time complexity is actually linear in the size of the grammar although it is n-EXPTIME in the largest arity of non-terminal symbols and the size of the property. Kobayashi and Ong [20] later extended the result to show that, for the full modal μ-calculus, the time complexity is polynomial in the size of the grammar. These results gave a hope that one might be able to construct a practical model checker (although, as discussed earlier, it is hopeless to use Kobayashi and Ong's algorithms [18, 20] directly), which lead us to the present work.

Other theoretical studies on recursion schemes include: the equi-expressivity between recursion schemes and (variants of) higher-order pushdown automata [11, 17] and the complexity result on the model checking for subclasses of modal μ-calculus [19].

Other verification techniques for functional programs Flow analysis, abstract interpretation, and type-based analysis have been used as standard techniques for analysis/verification of functional programs [23]. Those techniques (except dependent type systems, which require human intervention) are usually incomplete even for the simply-typed λ-calculus with recursion and finite data domains, for which our verification method is complete.

We are not aware of previous model checkers that can verify higher-order recursive functions in a sound and *complete* manner. Most of the existing software model checkers [5, 6] are based on either finite state or pushdown model checking; some approximation is necessary for encoding higher-order recursive functions into finite state or pushdown systems, so that the completeness is lost. For example, SLAM [5] deals with function pointers by replacing them with non-deterministic jumps to the functions they may point to.

Bakewell and Ghica [4] proposed a model checker called MAGE, based on game semantics. Although game semantics has been studied for higher-order functional languages, their model checking algorithm and implementation deal with neither higher-order functions nor recursion.

Inference of Intersection Types Since our model checking algorithm is a type inference algorithm for the intersection type system presented in Section 2.3, there may be some connection between our algorithm and type inference algorithms for intersection types [8, 9, 14, 27]. In particular, earlier algorithms for intersection type inference [9, 27] first finds a normal form, and then obtains a principal typing for the normal form; this seems somewhat similar to Steps 1 and 2 of our algorithm, which first reduce a given recursion scheme, and then extract type information. There are, however, several important differences. First, the intersection type inference algorithms aim to infer a principal typing, while our algorithm does not. Secondly, our type system is decidable, while the intersection type systems studied in the literature are usually undecidable (as the typability coincides with strong normalization). Thirdly (and most importantly), the intersection type systems studied in [8, 9, 14, 27] guarantee that typable terms (possibly with certain additionial conditions) have the strong normalization property, while the intersection type system for recursion schemes does not. That is why our algorithm is hybrid: type information is extracted after a finite number of reduction steps, and then another algorithm is used for deciding whether the recursion scheme is typable using extracted type information. Despite the differences above, it would be interesting to study the relationship between our algorithm and their algorithms in more detail, to see whether some of their techniques can be used for optimizing our type inference algorithm.

7. Conclusion

We have proposed a new model checking algorithm for recursion schemes, proved its correctness, and demonstrated its effectiveness through an implementation of the model checker.

The next important step of this line of research is to implement software model checkers for functional languages such as ML and Haskell on top of the model checker for recursion schemes. For that purpose, as outlined in [18], we need to combine the model checker with predicate abstractions and CEGAR (counter-example-guided abstract refinement). To construct practical software model checkers, we probably need to improve the efficiency of the underlying model checker for recursion schemes by several orders of magnitude. There are a number of potentially useful optimizations, such as a more tight integration of the three steps: for example, when Step 3 fails (i.e. if $S : q_0 \in \Gamma$ does not hold), we should be able to use the computed type environment Γ in Step 1 for effectively deciding which open node should be expanded (to get more type information). Future work also includes the optimization discussed in Section 5, and an extension of our algorithm to deal with the full modal μ-calculus (or equivalently, alternating parity tree automata), based on Kobayashi and Ong's type system [20]. Actually, it is not difficult to adapt our algorithm to deal with the full modal μ-calculus in a naive manner, but the resulting algorithm is probably too slow: finding an efficient model-checking algorithm for the full modal μ-calculus remains a challenge.

The current model checking algorithm is for verification of closed programs. It can, however, be easily adapted to deal with open programs (which take unknown function arguments as inputs), provided that either the specifications (i.e. intersection types) or typical code of the unknown functions are given. Integration with testing may be useful, as typical inputs are provided during testing.

Despite the promising experimental results of our model-checking algorithm, one cannot overcome the theoretical lower-bound of the worst-case complexity: $(n-1)$-EXPTIME hardness for deterministic trivial automata, and n-EXPTIME hardness for the full modal μ-calculus [24]. We conclude this paper with informal remarks on why the verification of higher-order functions is so hard. We think that the hardness of the verification of higher-order functions is strongly related to the capability of higher-order functions to express computation compactly. As we show in Appendix B, even without recursion, a word of length $O(\mathbf{exp}_n(|\mathcal{G}|))$ can be generated by a recursion scheme of order-n. Thus, for such a recursion scheme, an n-EXPTIME model-checking algorithm is no worse than a linear time algorithm for finite-state machines: since the corresponding finite-state machine must have $O(\mathbf{exp}_n(|\mathcal{G}|))$ states, the linear time algorithm is actually n-EXPTIME in the size of the recursion scheme. If this is indeed the primary reason for the hardness of the verification of higher-order functions, then the verification of a well-organized higher-order functional program can be actually easier than that of the corresponding while-program (that expresses the same computation); higher-order functions can make the high-level structure of a program explicit, which can be exploited by a verification algorithm. That may explain why our prototype model checker is reasonably fast in practice.

References

[1] Objective caml. `http://caml.inria.fr/ocaml/`.

[2] K. Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. *Logical Methods in Computer Science*, 3(3), 2007.

[3] 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.

[4] A. Bakewell and D. R. Ghica. On-the-fly techniques for game-based software model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 78–92. Springer-Verlag, 2008.

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

[6] 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.

[7] M. Blume, U. A. Acar, and W. Chae. Exception handlers as extensible cases. In *Proceedings of APLAS 2008*, volume 5356 of *LNCS*, pages 273–289. Springer-Verlag, 2008.

[8] G. Boudol. On strong normalization and type inference in the intersection type discipline. *Theor. Comput. Sci.*, 398(1-3):63–81, 2008.

[9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda-calculus semantics. In *Essays on Combinatory Logic, Lambda Calculus, and Foundation*, pages 535–560. Academic Press, 1980.

[10] B. Courcelle. The monadic second-order logic of graphs IX: machines and their behaviours. *Theoretical Computer Science*, 151:125–162, 1995.

[11] M. Hague, A. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 452–461. IEEE Computer Society, 2008.

[12] 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.

[13] 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.

[14] A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theor. Comput. Sci.*, 311(1-3):1–70, 2004.

[15] 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.

[16] 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.

[17] 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.

[18] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proc. of POPL*, pages 416–428, 2009.

[19] N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. In *Proceedings of ICALP 2009*, LNCS. Springer-Verlag, 2009.

[20] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of LICS 2009*. IEEE Computer Society Press, 2009.

[21] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Prog. Lang. Syst.*, 22(2):340–377, 2000.

[22] M. Might and O. Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *J. Funct. Program.*, 18(5-6):821–864, 2008.

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[24] 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.

[25] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Maths. Soc*, 141:1–35, 1969.

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

[27] S. R. D. Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theor. Comput. Sci.*, 28:151–169, 1984.

Appendix

A. Proofs

This appendix provides proofs omitted in the main text.

LEMMA A.1. *Let $\mathcal{C}$ be the fully-expanded configuration graph. If $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$, then the following conditions hold.*
(I) If $\mathcal{C}''$ is obtained from expansions of $\mathcal{C}'$, then $\mathcal{C}'' \preceq_{\pi,t} \mathcal{C}$.
(II) $\tau_{t,\mathcal{C}(\pi)} \in \mathit{Elim}(\tau_{t,\mathcal{C}'(\pi)})$.

Proof Let the label of $\mathcal{C}(\pi)$ be $\langle t\,\widetilde{s}, q, \texttt{closed}\rangle$. The proof proceeds by induction on the sort of t. If the sort of t is $\texttt{o}$, then by the definition of the relation $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$, we have $\tau_{t,\mathcal{C}'(\pi)} = q$. By the definitions of expansions and $\tau_{t,N}$, we have $\tau_{t,\mathcal{C}'(\pi)} = \tau_{t,\mathcal{C}''(\pi)} = \tau_{t,\mathcal{C}(\pi)} = q$. Thus, we have $\mathcal{C}'' \preceq_{\pi,t} \mathcal{C}$ and $\mathit{Elim}(\tau_{t,\mathcal{C}'(\pi)}) = \{q\} \ni \tau_{t,\mathcal{C}(\pi)}$ as required.

If the sort of t is $\kappa_1 \to \kappa_2$, then by the condition $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$, we have:
(i) $\widetilde{s} = s_0\widetilde{s}'$;
(ii) $\tau_{t,\mathcal{C}(\pi)} = \bigwedge_{i=1}^{m} \tau_i \to \tau$;
(iii) $\mathcal{C}' \preceq_{\pi,ts_0} \mathcal{C}$; and
(iv) for each τ_i, there exists π_i such that $\tau_{s_0,\mathcal{C}(\pi\pi_i)} = \tau_i$ and $\mathcal{C}' \preceq_{\pi\pi_i,s_0} \mathcal{C}$.
By the induction hypothesis (I), we have $\mathcal{C}'' \preceq_{\pi,ts_0} \mathcal{C}$ and $\mathcal{C}'' \preceq_{\pi\pi_i,s_0} \mathcal{C}$, which implies (I). By the induction hypothesis (II), we also have $\tau_{ts_0,\mathcal{C}(\pi)} \in \mathit{Elim}(\tau_{ts_0,\mathcal{C}'(\pi)})$ and $\tau_{s_0,\mathcal{C}(\pi\pi_i)} \in \mathit{Elim}(\tau_{s_0,\mathcal{C}'(\pi\pi_i)})$. By the definition of $\tau_{t,\mathcal{C}'(\pi)}$, it is of the form:

$$\bigwedge\{\tau_{s_0,\mathcal{C}'(\pi\pi_1)},\ldots,\tau_{s_0,\mathcal{C}'(\pi\pi_m)},\sigma_1,\ldots,\sigma_k\} \to \tau_{ts_0,\mathcal{C}'(\pi)}.$$

By the definition of Elim, we can construct $\tau_{t,\mathcal{C}(\pi)}$ as an element of $\mathit{Elim}(\tau_{t,\mathcal{C}'(\pi)})$ as follows:
(i) choose τ_i from $\mathit{Elim}'(\tau_{s_0,\mathcal{C}'(\pi\pi_i)})$; and
(ii) from $\mathit{Elim}'(\sigma_j)$, choose $\top$ when σ_j contains a type variable; otherwise choose σ_j (in which case we have $\sigma_j \in \{\tau_1,\ldots,\tau_m\}$). Thus, we have $\tau_{t,\mathcal{C}(\pi)} \in \mathit{Elim}(\tau_{t,\mathcal{C}'(\pi)})$ as required. □

LEMMA A.2. *Let $N = \mathcal{C}(\pi)$ be a node of a closed configuration graph $\mathcal{C}$, and suppose that N is labeled with $\langle t\,\widetilde{s}, q, \texttt{closed}\rangle$. Then, there exists a finitely-expanded graph $\mathcal{C}'$ such that $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$.*

Proof The proof proceeds by induction on the sort of t. If the sort is $\texttt{o}$, then the result follows immediately: just expand the initial graph until N is expanded, and let $\mathcal{C}'$ be the resulting graph.

If the sort is $\kappa_1 \to \kappa_2$, then $\tau_{t,N}$ is of the form $\bigwedge_{i=1}^{m} \tau_i \to \tau$, and $\widetilde{s} = s_0\widetilde{s}'$. By the induction hypothesis, there exists a finitely expanded graph $\mathcal{C}'_0$ such that $\mathcal{C}'_0 \preceq_{\pi,ts_0} \mathcal{C}$. By the definition of $\tau_{t,N}$, for each $i \in \{1,\ldots,m\}$, there exists π_i such that $\tau_{s_0,\mathcal{C}(\pi\pi_i)} = \tau_i$. By the induction hypothesis, there exists a finitely expanded graph $\mathcal{C}'_i$ such that $\mathcal{C}'_i \preceq_{\pi\pi_i,s_0} \mathcal{C}$. Thus, the union of $\mathcal{C}'_0, \mathcal{C}'_1, \ldots, \mathcal{C}'_m$ satisfies the required condition. □

B. On the Expressive Power of Recursion Schemes

This section demonstrates that a recursion scheme can generate a very large word or tree, even without using recursion.

The following order-1 recursion scheme generates a word $\texttt{a}^{2^m}\texttt{c}$.

$$\begin{array}{l}
S \to F_0\,\texttt{c}\\
F_0\,x \to F_1(F_1\,x)\\
F_1\,x \to F_2(F_2\,x)\\
\cdots\\
F_{m-1}\,x \to F_m(F_m\,x)\\
F_m\,x \to \texttt{a}\,x
\end{array}$$

Note that S can be reduced to $F_m^{2^m}\,\texttt{c}$, and then to $\texttt{a}^{2^m}\,\texttt{c}$.

The following order-2 recursion scheme generates a word $\texttt{a}^{2^{2^m}}\texttt{c}$.

$$\begin{array}{l}
S \to F_0\,\texttt{a}\,\texttt{c}\\
F_0\,x\,y \to F_1(F_1\,x)\,y\\
F_1\,x\,y \to F_2(F_2\,x)\,y\\
\cdots\\
F_{m-1}\,x\,y \to F_m(F_m\,x)\,y\\
F_m\,x\,y \to x(x\,y)
\end{array}$$

S can be reduced to $F_m^{2^m}\,\texttt{a}\,\texttt{c}$, and then to $\texttt{a}^{2^{2^m}}\,\texttt{c}$.

The following order-3 recursion scheme generates a word $\texttt{a}^{2^{2^{2^m}}}\texttt{c}$.

$$\begin{array}{l}
S \to F_0\,\mathit{Twice}\,\texttt{a}\,\texttt{c}\\
F_0\,x\,y\,z \to F_1(F_1\,x)\,y\,z\\
F_1\,x\,y\,z \to F_2(F_2\,x)\,y\,z\\
\cdots\\
F_{m-1}\,x\,y\,z \to F_m(F_m\,x)\,y\,z\\
F_m\,x\,y\,z \to x(x\,y)\,z\\
\mathit{Twice}\,y\,z \to y(y\,z)
\end{array}$$

S can be reduced to $F_m^{2^m}\,\mathit{Twice}\,\texttt{a}\,\texttt{c}$, and then to $(\mathit{Twice}^{2^{2^m}}\,\texttt{a})\texttt{c}$, which is further reduced to $\texttt{a}^{2^{2^{2^m}}}\,\texttt{c}$. By repeating the same construction (i.e. by replacing $\texttt{a}$ with *Twice*), we can construct an order-n recursion scheme (without recursion) that generates the word $\texttt{a}^{\mathbf{exp}_n(m)}\texttt{c}$.