Model Checking Higher-Order Programs[1]

Naoki Kobayashi
The University of Tokyo

We propose a novel verification method for higher-order functional programs based on higher-order model checking, or more precisely, model checking of higher-order recursion schemes (recursion schemes, for short). The most distinguishing feature of our verification method for higher-order programs is that it is sound, complete, and automatic for the simply-typed λ-calculus with recursion and finite base types, and for various program verification problems such as reachability, flow analysis, and resource usage verification. We first show that a variety of program verification problems can be reduced to model checking problems for recursion schemes, by transforming a program into a recursion scheme that generates a tree representing all the interesting possible event sequences of the program. We then develop a new type-based model checking algorithm for recursion schemes and implement a prototype recursion scheme model checker. To our knowledge, this is the first implementation of a recursion scheme model checker. Experiments show that our model checker is reasonably fast, despite the worst-case time complexity of recursion scheme model checking being hyper-exponential in general. Altogether, the results provide a new, promising approach to verification of higher-order functional programs.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Verification

Additional Key Words and Phrases: Type systems, model checking, higher-order recursion schemes

1. INTRODUCTION

Program verification techniques have been studied extensively, due to the increasing importance of software reliability. There are still limitations in the current verification technology, however. Software model checking [Ball et al. 2001; Ball and Rajamani 2002; Beyer et al. 2007] has been mainly applied to imperative programs with first-order procedures, and applications to higher-order programs with arbitrary recursion have been limited. For higher-order programs, *type systems* have been recognized as effective techniques for program verification. However, they often require explicit type annotations (as in dependent type systems), or suffer from many false alarms. For example, ML type system [Damas and Milner 1982] allows automated type inference, but rejects many type-safe programs.

This article proposes a novel verification technique for higher-order programs. The most distinguishing feature of our new technique is that it is sound, *complete*,

This is a revised and extended version of the paper that appeared in Proceedings of POPL 2009 under the title "Types and Higher-Order Recursion Schemes for Verification of Higher-Order Programs" and the paper that appeared in Proceedings of PPDP2009 under the title "Model-Checking Higher-Order Functions".

and *fully automatic* for the class of programs written in the simply-typed λ-calculus with recursion and finite base types, and for a variety of verification problems, including reachability ("Does a given program reach a program point **fail**?"), flow analysis ("Does a subterm e of a given program evaluate to a value generated at program point l?"), and resource usage verification ("Does a given program access resources such as files and memory in a valid manner?").

Our verification technique is based on model checking of *higher-order recursion schemes* (recursion schemes, for short) [Ong 2006], and consists of two main steps. In the first step, a program verification problem is reduced to a problem of recursion scheme model checking. In the second step, the problem of recursion scheme model checking is solved by reduction to a type checking problem. Thus, our verification method can be considered an integration of two main previous approaches to program verification: model checking and type systems. A good consequence of the integration is that as in ordinary type-based program verification, our method can generate types as certificates when a given property is satisfied, and as in ordinary model checking, it can generate an error path as a counter-example when a property is violated. In the rest of this section, we first introduce higher-order recursion schemes informally, and then give an overview of the two steps mentioned above.

1.1 Higher-Order Recursion Schemes

A higher-order recursion scheme [Knapik et al. 2002; Ong 2006] is a kind of tree grammar for generating a single, possibly infinite, ranked tree. From a programming language point of view, a recursion scheme may be regarded as a term of the simply-typed λ-calculus with recursion and tree constructors (but without destructors). A recursion scheme of order 0 is just a (deterministic) regular tree grammar that generates a regular tree. For example, consider the following order-0 recursion scheme:

$$S \to \mathtt{a}\, F \qquad F \to \mathtt{b}\, S\, S$$

Here, the capitalized symbols S and F denote non-terminals, and the non-capitalized symbols $\mathtt{a}$ and $\mathtt{b}$ represent terminals (or tree constructors). The term $\mathtt{b}\, S\, S$ should be interpreted as the tree whose root is labeled with $\mathtt{b}$, having as children two leaves labeled with S. The start symbol S is rewritten as follows (where the redexes are underlined).

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

Figure 1(a) shows the tree generated by the recursion scheme. In the case of *higher-oder* recursion schemes, each non-terminal can take trees or higher-order functions on trees (depending on the order of recursion schemes) as parameters. The parameters must be trees in order-1 recursion schemes, while they can be functions on trees (i.e. functions that take trees as input and return trees) in order 2, and functions on functions on trees (i.e. functions that take functions on trees as input, and return trees) in order 3. The following is an example of order-1 recursion scheme:

$$S \to F\ \mathtt{c} \qquad\qquad F\ x \to \mathtt{br}\ x\ (\mathtt{a}(F(\mathtt{b}(x))))$$

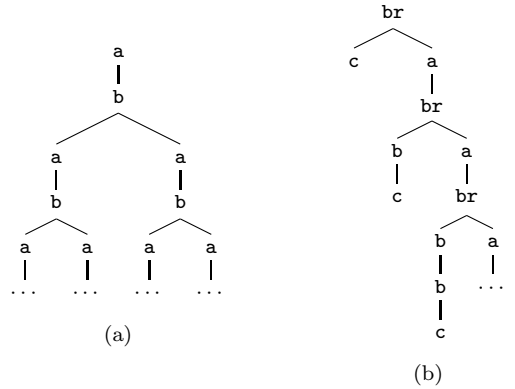


Fig. 1. The trees generated by recursion schemes

Here, the non-terminal F takes a tree parameter x. The start symbol S is rewritten as follows.

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

The tree generated by the recursion scheme is shown in Figure 1(b). With the label `br` being ignored, the tree consists of paths labeled by $\mathtt{a}^n\mathtt{b}^n\mathtt{c}$.

As the example above indicates, the trees generated by recursion schemes are in general more complex than regular trees. Ong [2006], however, proved that the modal μ-calculus model checking of the tree generated by a recursion scheme (namely, the decision problem: "Given a recursion scheme $\mathcal{G}$ and a modal μ-calculus formula φ, does the tree generated by $\mathcal{G}$ satisfy φ?") is decidable.

The model checking of higher-order recursion schemes has the following attractive features. First, the class of recursion schemes is, to date, the most expressive (generically-defined) class of infinite trees for which the modal μ-calculus model checking problem is decidable. It subsumes the classes of regular trees, algebraic trees [Courcelle 1983], and the trees generated by higher-order pushdown automata [Knapik et al. 2002]. Secondly, recursion schemes are *high-level* descriptions of infinite trees. They are essentially typed higher-order functional programs with recursion and tree constructors, so that it is easy to model higher-order programs by recursion schemes.

Recursion scheme model checking subsumes finite state model checking [Clarke et al. 1999] and pushdown model checking [Schwoon 2002]. Indeed, model checking of order-0 recursion schemes corresponds to finite state model checking: see Remark 2.3. Similarly, model checking of order-1 recursion schemes corresponds to pushdown model checking.

1.2 From Higher-Order Program Verification to Recursion Scheme Model Checking

In program verification, we are often interested in temporal properties and/or outputs of a program. For example, typical verification problems are: (i) "Can an assertion failure occur?", (ii) "Is an opened file eventually closed?", (iii) "Can function g be called from a program point l?", (iv) "Does function f always return a

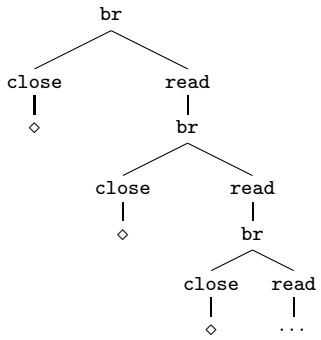


Fig. 2. A tree that represents possible resource access sequences

certain well-formed data structure?", etc. The key idea of our verification method is to transform a program into a recursion scheme that generates a tree representing all the possible event sequences or outputs of interest.

For example, consider the following program (written in OCaml-like language), where `_` represents a non-deterministic boolean.

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

The program first defines a recursion function `g`, which takes a file pointer x as an argument, and accesses x recursively. It then opens a read-only file `foo`, and then passes a file pointer to `g`. Suppose that we wish to verify that the file `foo` is accessed according to the specification $\texttt{read}^*\texttt{close}$. We shall transform the above program into a recursion scheme that generates a tree like the one in Figure 2, which shows how the file is accessed by the program. In the figure, `br` and $\diamond$ denote a non-deterministic branch and a program termination respectively. Once we have obtained such a recursion scheme, it is sufficient to model-check the recursion scheme to verify that every finite path of the tree generated by the recursion scheme is labeled by $\texttt{read}^*\texttt{close}$ (with `br` ignored), and that every infinite path is labeled by a prefix of $\texttt{read}^*\texttt{close}$.[2]

The tree in Figure 2 can be generated by the following recursion scheme.

$$\begin{array}{rcl} G\,x\,k & \to & \texttt{br}\,(\texttt{close}\,k)\,(\texttt{read}\,(G\,x\,k)) \\ S & \to & G\,\texttt{d}\,\diamond \end{array}$$

Notice the similarity of this recursion scheme to the source program. The first rule corresponds to the definition of function `g`, and the second rule corresponds to the call of function `g` (with `d` being a dummy argument, which is not important here; we will see a meaningful representation of file pointers in Section 3). The non-terminal G corresponds to function `g`, but it takes an additional parameter k, which represents how the file `foo` will be accessed *after* the call of G. The non-terminal `br` corresponds to the conditional, with the part "$\texttt{close}\,k$" (meaning that

[2]Here, we do not check termination of the program, so there may be an infinite path that does not contain `close`.

the file is closed and then accessed according to k) corresponding to the then-part, and the part "`read` $(G\,x\,k)$" to the else-part. Thus, the recursion scheme above is essentially a continuation-passing-style (CPS) representation of the definition of `g`.

As we will see in Section 3, CPS transformation [Plotkin 1975; Danvy and Filinski 1992; Appel 1992] with some additional tricks is indeed sufficient for systematically constructing a recursion scheme that generates all the possible event sequences. Thus, program verification problems can be automatically transformed to problems of recursion scheme model checking.

1.3 Type-Based Model Checking of Recursion Schemes

Given that verification problems for higher-order programs can be reduced to recursion scheme model checking, an important question is whether there is an efficient algorithm to solve the model checking problems. There are a few existing model checking algorithms for recursion schemes: Ong's algorithm based on variable profiles [Ong 2006], and Hague et al.'s algorithm based on reduction to model checking of collapsible pushdown automata [Hague et al. 2008]. Aehlig [Aehlig et al. 2005] has also developed a model checking algorithm for recursion schemes for a restricted class of properties, described by so called *trivial automata* (Büchi tree automata where all the states are accepting states), which are sufficient for the verification problems considered in this article.

All the algorithms above are, however, mainly of theoretical interest, and cannot be directly applied in practice. The problem is that model checking of order-k higher-order recursion schemes is k-EXPTIME complete in general [Ong 2006], and that all the above algorithms *almost always* suffer from this k-EXPTIME bottleneck.

We thus develop a new model checking algorithm for recursion schemes, which is based on *types*. We use tree automata (more precisely, the class of trivial automata considered by Aehlig et al. [2005]) instead of modal μ-calculus formulas to express tree properties.[3] We then regard an automaton state q as the type of trees, accepted by the automaton from state q. A function type $q_1 \to q_2$ is then interpreted as functions that take a tree of type q_1 and return that of type q_2. An intersection type $(q_1 \to q_2) \wedge (q_3 \to q_4)$ is interpreted as functions that return a tree of type q_2 if the argument has type q_1, and return a tree of type q_4 if the argument has type q_3. Based on this intuition, one can naturally construct an intersection type system that is equivalent to model checking, i.e., a type system parameterized by an automaton $\mathcal{B}$, such that a recursion scheme is well-typed if, and only if, the tree generated by the recursion scheme is accepted by $\mathcal{B}$. This connection allows us to reduce the recursion scheme model checking to the problem of type checking.

The type checking problem can be solved by using a standard fixed-point computation algorithm. It suffices to enumerate all the possible types of non-terminal symbols, and filter out invalid types until the fixed-point computation converges. This algorithm has a few pleasant properties. First, it is very simple, so that it is easy to prove the correctness of the algorithm (hence also easy to prove the de-

[3]The class of trivial automata is more restricted in terms of the expressive power of tree properties than the class of modal μ-calculus formulas. The extension to deal with the full modal μ-calculus model checking is discussed elsewhere [Kobayashi and Ong 2009].

cidability of recursion scheme model checking). Secondly, the algorithm is *efficient* from the viewpoint of complexity theory. It runs in time *linear* in the size of the recursion scheme under the assumption that the largest size of the types (or sorts, in the terminology of the present paper) of symbols and the specification size (i.e. the size of the automaton) are fixed, although it is still k-EXPTIME in the size of the automaton and the largest arity of symbols.

The simple algorithm mentioned above is, however, still too inefficient to be applied in practice. The problem is that, although the algorithm runs in time linear in the size of a recursion scheme, the constant factor is *huge*. It is hyper-exponential in the specification size and the largest arity, and can be as large as $10^{10000000000000000000}$ even for order-3 recursion schemes. This is due to the fact that the number of possible types of non-terminal symbols blows up very quickly with an increase of the order of recursion schemes, and the algorithm must search valid types of non-terminal symbols from such a huge set of candidates.

We thus develop a more sophisticated, hybrid algorithm, which runs much faster for realistic inputs. As mentioned above, the main problem of the simple algorithm is that the number of possible types of non-terminals is too large. The hybrid algorithm obtains a much smaller set of candidates for the types of non-terminals, by reducing a recursion scheme a finite number of steps and inspecting how each non-terminal is used in the reduction. The algorithm then searches valid typings from the set of candidate types by using a fixed-point computation. This makes the search for valid typings terminate very quickly. The price to pay instead is that completeness may be lost. By iteratively increasing the number of reduction steps, however, we can actually guarantee the completeness of the algorithm; The algorithm eventually terminates, and it reports either a success of verification or a property violation correctly.

We have implemented a recursion scheme model checker based on the hybrid algorithm above. According to experiments, our model checker is reasonably fast, even though the recursion scheme model checking is hyper-exponential in general.

1.4 Contributions and Overview of Article

Contributions of this article are summarized as follows.

(1) A new program verification method based on reductions of program verification problems to recursion scheme model checking. The method is sound and complete for various verification problems for the simply-typed λ-calculus with recursion and finite base types.

(2) Reduction of recursion scheme model checking to typability in an intersection type system. The reduction yields a very simple algorithm for recursion scheme model checking, which runs in time linear in the size of recursion schemes if certain parameters are fixed.

(3) The first realistic algorithm for recursion scheme model checking. The algorithm is based on a novel idea of inferring intersection types in an on-demand manner.

(4) The first implementation of a recursion scheme model checker and experiments. Despite the extremely high worst-case complexity, the model checker shows

good performance for typical inputs, obtained from small but tricky program verification problems.

This article is the archival version of two conference papers: [Kobayashi 2009b] and [Kobayashi 2009a]. The first two contributions were first reported in [Kobayashi 2009b], and the latter two were reported in [Kobayashi 2009a]. From the conference versions, we have polished various definitions and proofs, and expanded examples, discussions, proofs, and experiments.

The rest of this article is structured as follows. Section 2 reviews the formal definitions of higher-order recursion schemes and the model checking problem for recursion schemes. Section 3 shows reductions from program verification problems to recursion scheme model checking. Section 4 reduces recursion scheme model checking to a type checking problem, and presents a naive model checking algorithm based on the reduction. Section 5 presents a more practical model checking algorithm. Section 6 reports the implementation of a prototype recursion scheme model checker and experimental results. Section 7 discusses related work. Section 8 concludes this article, with discussion on future perspectives.

2. PRELIMINARIES

This section reviews the definition of higher-order recursion schemes and model checking problems.

NOTATION 2.1. *We write $\widetilde{v}$ for a possibly empty sequence $v_1, \ldots, v_n$. We write $dom(f)$ for the domain of a map f. A map is represented by a set of bindings of the form $x \mapsto v$: $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ denotes the map f such that $dom(f) = \{x_1, \ldots, x_n\}$ and $f(x_i) = v_i$ for $i \in \{1, \ldots, n\}$. When $dom(f) \cap dom(g) = \emptyset$, $f \cup g$ denotes the map h such that $dom(h) = dom(f) \cup dom(g)$ with $\forall x \in dom(f).h(x) = f(x)$ and $\forall x \in dom(g).h(x) = g(x)$. We often omit $\cup$ and write $f\{x \mapsto v\}$ for $f \cup \{x \mapsto v\}$. For readability, we sometimes write $\{x_1 : v_1, \ldots, x_n : v_n\}$ for $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ (especially when v_i contains the symbol '$\rightarrow$').*

We write $\mathbf{N}_+$ for the set of positive integers. For a set S, S^* denotes the set of sequences of elements of S.

We first define (possibly infinite, labeled) trees.

Definition 2.1 *(ranked alphabet, trees).* A *ranked alphabet* is a map from a finite set of symbols to non-negative integers. When Σ is a ranked alphabet, $\Sigma(a)$ is called the arity of a. A non-empty subset S of $\mathbf{N}_+^*$ is called a *tree* if $\{\pi, \pi 1, \ldots, \pi(j-1)\} \subseteq S$ whenever $\pi j \in S$. An *L-labeled tree* is a map from a tree to L. Let Σ be a ranked alphabet, and L be $dom(\Sigma)$. An L-labeled tree T is called a *ranked Σ-labeled tree* if, whenever $T(\pi) = a$, $\{i \mid \pi i \in dom(T)\} = \{1, \ldots, \Sigma(a)\}$. Let $\bot$ be a special symbol of arity 0. For a ranked alphabet Σ with $\bot \notin dom(\Sigma)$, we write $\Sigma^\bot$ for the ranked alphabet $\Sigma\{\bot \mapsto 0\}$. We define the binary relation $\sqsubseteq$ on ranked $\Sigma^\bot$-labeled trees by: $T_1 \sqsubseteq T_2$ iff (i) $dom(T_1) \subseteq dom(T_2)$ and (ii) $\forall \pi \in dom(T_1).T_1(\pi) = T_2(\pi) \vee T_1(\pi) = \bot$. For a set S of ranked $\Sigma^\bot$-labeled trees, we write $\bigsqcup S$ for the least upper-bound of S with respect to $\sqsubseteq$ if it exists.

We often write a for the labeled tree $\{\epsilon \mapsto a\}$. When $T_1, \ldots, T_n$ are labeled trees, we write $a\, T_1 \ldots T_n$ for the labeled tree T such that $dom(T) = \{\epsilon\} \cup \{i\pi \mid \pi \in dom(T_i), i \in \{1, \ldots, n\}\}$ with $T(\epsilon) = a$ and $T(i\pi) = T_i(\pi)$.

Definition 2.2 *(sorts).* The set of *sorts*, ranged over by κ, is inductively defined by:

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

The *order* and *arity* of κ, written $\mathit{order}(\kappa)$ and $\mathit{arity}(\kappa)$ respectively, are defined by:

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

Intuitively, 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.

Definition 2.3 *(applicative terms).* Let Σ be a ranked alphabet. The set of applicative terms is defined by:

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

where a ranges over $\mathit{dom}(\Sigma)$ and x ranges over a set of variables. A *sort environment*, written $\mathcal{K}$, is a map from variables to sorts. The sort assignment relation $\mathcal{K} \vdash_\Sigma t : \kappa$ is the least relation closed under the following rules:

$$\mathcal{K} \vdash_\Sigma a : \underbrace{\mathsf{o} \to \cdots \to \mathsf{o}}_{\Sigma(a)} \to \mathsf{o}$$

$$\mathcal{K} \cup \{x \mapsto \kappa\} \vdash_\Sigma x : \kappa$$

$$\frac{\mathcal{K} \vdash_\Sigma t_1 : \kappa_2 \to \kappa_1 \qquad \mathcal{K} \vdash_\Sigma t_2 : \kappa_2}{\mathcal{K} \vdash_\Sigma t_1 t_2 : \kappa_1}$$

When $\mathcal{K} \vdash_\Sigma t : \kappa$ holds, we say that t *has sort* κ *under* $\mathcal{K}$.

We now review the definition of the syntax and the rewriting relation for higher-order recursion schemes [Ong 2006].

Definition 2.4 *(higher-order recursion schemes).* A (deterministic) higher-order recursion scheme, written $\mathcal{G}$, is a quadruple $(\Sigma, \mathcal{N}, \mathcal{R}, S)$, where

(1) Σ is a *ranked alphabet*. The elements of $\mathit{dom}(\Sigma)$ are called terminals.
(2) $\mathcal{N}$ is a map from a finite set of symbols called *non-terminals* to sorts. $\mathit{dom}(\Sigma)$ and $\mathit{dom}(\mathcal{N})$ must be disjoint.
(3) $\mathcal{R}$ is a map from the set of non-terminals (i.e. $\mathit{dom}(\mathcal{N})$) to terms of the form $\lambda x_1. \cdots \lambda x_\ell.t$, where (i) t is an applicative term, (ii) $\mathcal{N}(F)$ is of the form $\kappa_1 \to \cdots \to \kappa_\ell \to \mathsf{o}$, and (iii) $\mathcal{N}\{x_1 \mapsto \kappa_1, \ldots, x_\ell \mapsto \kappa_\ell\} \vdash_\Sigma t : \mathsf{o}$ (where non-terminals are treated as variables).
(4) S is a non-terminal called the *start symbol*. We require that $S \in \mathit{dom}(\mathcal{N})$ and $\mathcal{N}(S) = \mathsf{o}$.

The *order* of a non-terminal F, written $\mathit{order}(F)$, is the order of its sort, i.e. $\mathit{order}(\mathcal{N}(F))$. The *order* of a recursion scheme $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$, written $\mathit{order}(\mathcal{G})$, is the highest order of its non-terminals, i.e. $\mathit{max}(\{\mathit{order}(F) \mid F \in \mathit{dom}(\mathcal{N})\})$.

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

(i) $F\,s_1 \cdots s_\ell \longrightarrow_{\mathcal{G}} [s_1/x_1, \ldots, s_\ell/x_\ell]t$ if $\mathcal{R}(F) = \lambda x_1.\cdots\lambda x_\ell.t$ with t being an applicative term.

(ii) If $t_i \longrightarrow_{\mathcal{G}} t_i'$, then $a\,t_1 \cdots t_\ell \longrightarrow_{\mathcal{G}} a\,t_1 \cdots t_{i-1}\,t_i'\,t_{i+1} \cdots t_\ell$.

Here, ℓ may be 0, and $[s_1/x_1, \ldots, s_\ell/x_\ell]t$ is the term obtained by replacing each occurrence of $x_i (1 \leq i \leq \ell)$ with s_i. We omit the subscript $\mathcal{G}$ whenever it is clear from the context.

We review the definition of the *value tree* [Ong 2006], i.e., the (possibly infinite) tree represented by a higher-order recursion scheme. Intuitively, the value tree is the (possibly infinite) $\Sigma^\perp$-labeled ranked tree obtained by infinitary, fair rewriting of the start symbol S, where a rewriting sequence is *fair* if every redex is eventually reduced. More formally, $[\![\mathcal{G}]\!]$ is defined as follows.

Definition 2.5 *(value trees).* For an applicative term t, we define $t^\perp$ inductively by (i) $(Fs_1 \cdots s_n)^\perp = \perp$ and (ii) $(as_1 \cdots s_n)^\perp = a({s_1}^\perp) \cdots ({s_n}^\perp)$ (where $n \geq 0$). The *value tree* of $\mathcal{G}$, written $[\![\mathcal{G}]\!]$, is $\bigsqcup\{t^\perp \mid S \longrightarrow_{\mathcal{G}}^* t\}$.

In the above definition, $\bigsqcup\{t^\perp \mid S \longrightarrow_{\mathcal{G}}^* t\}$ is well-defined, as the reduction relation is confluent. Note also that if t is a well-sorted applicative term, $t^\perp$ is a ranked $\Sigma^\perp$-labeled finite tree. Thus, $[\![\mathcal{G}]\!]$ is a ranked $\Sigma^\perp$-labeled tree.

NOTATION 2.2. *We often write:*

$$\mathcal{R} = \{F_1\,x_{1,1} \cdots x_{1,k_1} \to t_1, \quad \ldots, \quad F_m\,x_{m,1} \cdots x_{m,k_m} \to t_m\}$$

when $dom(\mathcal{R}) = \{F_1, \ldots, F_m\}$ *and* $\mathcal{R}(F_1) = \lambda x_{1,1}.\cdots\lambda x_{1,k_1}.t_1, \ldots, \mathcal{R}(F_m) = \lambda x_{m,1}.\cdots\lambda x_{m,k_m}.t_m$.

EXAMPLE 2.1. The example of order-1 recursion scheme given in Section 1.1 is formally a quadruple $\mathcal{G}_0 = (\Sigma, \mathcal{N}, \mathcal{R}, S)$, where:

$$\begin{array}{l}\Sigma = \{\mathtt{br} \mapsto 2, \mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0\}\\ \mathcal{N} = \{S : \mathtt{o}, F : \mathtt{o} \to \mathtt{o}\}\\ \mathcal{R} = \{S \mapsto F\ \mathtt{c}, F \mapsto \lambda x.(\mathtt{br}\ x\ (\mathtt{a}(F(\mathtt{b}(x)))))\}\end{array}$$

□

EXAMPLE 2.2. Let $\mathcal{G}_1$ be an order-2 recursion scheme $(\Sigma, \mathcal{N}, \mathcal{R}, S)$ where:

$$\begin{array}{l}\Sigma = \{\mathtt{br} \mapsto 2, \mathtt{a} \mapsto 1, \mathtt{c} \mapsto 0\}\\ \mathcal{N} = \{S : \mathtt{o}, F : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}, G : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}\}\\ \mathcal{R} = \{S \mapsto G\,\mathtt{a}, F \mapsto \lambda f.\lambda x.f(f\,x), G \mapsto \lambda f.(\mathtt{br}\,(f\,\mathtt{c})\,(G\,(F\,f)))\}\end{array}$$

The start symbol S is reduced as follows.

$$\begin{array}{l}\underline{S} \longrightarrow \underline{G\,\mathtt{a}} \longrightarrow \mathtt{br}\,(\mathtt{a}\,\mathtt{c})\,(\underline{G\,(F\,\mathtt{a})}) \longrightarrow \mathtt{br}\,(\mathtt{a}\,\mathtt{c})\,(\mathtt{br}\,(\underline{(F\,\mathtt{a}\,\mathtt{c})})\,(G\,(F\,(F\,\mathtt{a})))))\\ \longrightarrow \mathtt{br}\,(\mathtt{a}\,\mathtt{c})\,(\mathtt{br}\,(\mathtt{a}\,(\mathtt{a}\,\mathtt{c}))\,\overline{(G\,(F\,(F\,\mathtt{a})))}) \longrightarrow \cdots\end{array}$$

Since $(F^m\,\mathtt{a})\mathtt{c}$ reduces to $\mathtt{a}^{2^m}\,\mathtt{c}$ (where $\mathtt{a}^k\,\mathtt{c}$ represents $\underbrace{\mathtt{a}(\cdots(\mathtt{a}}_{k}\,\mathtt{c})\cdots)$), the tree generated by $\mathcal{G}_1$ consists of paths labeled by $\mathtt{br}^m\mathtt{a}^{2^{m-1}}\mathtt{c} (m \geq 1)$. □

Ong [2006] showed the decidability of the modal μ-calculus model checking of recursion schemes.

THEOREM 2.1 (ONG [2006]). *The model checking problem: "Given an order-k recursion scheme $\mathcal{G}$, and a modal μ-calculus formula ψ, does $[\![\mathcal{G}]\!]$ satisfy ψ?" is k-EXPTIME complete.*

Here, k-EXPTIME means DTIME($\mathbf{exp}_k(p(x))$) where x is the input size, $p(x)$ is a polynomial of x, and $\mathbf{exp}_k(x)$ is defined by $\mathbf{exp}_0(x) = x$ and $\mathbf{exp}_{i+1}(x) = 2^{\mathbf{exp}_i(x)}$.

In this article, we use tree automata for describing tree properties, instead of modal μ-calculus formulas, as that is convenient for discussing the model checking algorithm. It is well known that alternating parity tree automata have the same expressive power as the modal μ-calculus, and there are linear translations between the two [Emerson and Jutla 1991; Thomas 1997; Kupferman et al. 2000]. In this article, we use the following, more restricted class of tree automata called *deterministic trivial automata*, which are deterministic Büchi tree automata where all the states are final.

Definition 2.6 *(trivial automaton).* A *deterministic trivial automaton* $\mathcal{B}$ is a quadruple:

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

where Σ is a ranked alphabet, Q is a set of states, $q_0 \in Q$ is the initial state, and 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 $k = \Sigma(a)$. A Σ-labeled ranked tree T is *accepted* by $\mathcal{B}$ if there is a Q-labeled tree R such that (i) $dom(T) = dom(R)$; (ii) $R(\epsilon) = q_0$; and (iii) for every $x \in dom(R)$, $\delta(R(x), T(x)) = R(x1) \cdots R(xm)$ where $m = \Sigma(T(x))$. R is called a *run tree* of $\mathcal{B}$ over T.

For a trivial automaton $\mathcal{B} = (\Sigma, Q, \delta, q_0)$ where $\bot \notin dom(\Sigma)$, we write $\mathcal{B}^\bot$ for the trivial automaton $\mathcal{B} = (\Sigma^\bot, Q, \delta \cup \{(q, \bot) \mapsto \epsilon \mid q \in Q\}, q_0)$.

Aehlig [2007] considered non-deterministic Büchi tree automata where all the states are final, and called them *trivial automata*. In this article, we consider only deterministic trivial automata, so that we often omit the adjective "deterministic" and use the word *trivial automata* for deterministic trivial automata.

EXAMPLE 2.3. Consider the automaton $\mathcal{B}_0 = (\Sigma, \{q_0, q_1\}, \delta, q_0)$ where

$$\begin{array}{l}\Sigma = \{\mathtt{br} \mapsto 2, \mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0\}\\ \delta(q_0, \mathtt{br}) = q_0 q_0 \qquad \delta(q_1, \mathtt{br}) = q_1 q_1\\ \delta(q_0, \mathtt{a}) = q_0 \qquad \delta(q_0, \mathtt{b}) = \delta(q_1, \mathtt{b}) = q_1 \qquad \delta(q_0, \mathtt{c}) = \delta(q_1, \mathtt{c}) = \epsilon\end{array}$$

$\mathcal{B}_0$ accepts Σ-labeled trees whose paths are labeled by elements of $a^\omega + a^*b^\omega + a^*b^*c$, with `br` being ignored. The run tree of $\mathcal{B}_0$ over the tree generated by the recursion scheme in Example 2.1 (i.e. the tree shown in Figure 1(b)) is shown in Figure 3. □

The following is an immediate corollary of Ong's result (Theorem 2.1).

COROLLARY 2.2. *Given a recursion scheme $\mathcal{G}$ and a trivial automaton $\mathcal{B}$, it is decidable whether $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^\bot$.*

Henceforth, *recursion scheme model checking* refers to this restricted class of model checking problems. As discussed in Section 3, the restricted problems are sufficient for verification of safety properties of programs.

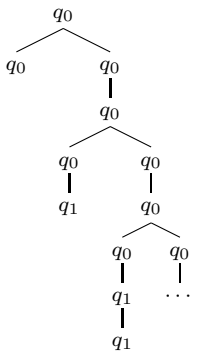


Fig. 3. A run tree of $\mathcal{B}_0$ for the tree in Figure 1(b) (see Example 2.3)

Remark 2.1. The recursion scheme model checking problem for deterministic trivial automata is $(k-1)$-EXPTIME complete. The upper bound comes from the facts that the complement of the language of trees accepted by a deterministic trivial automaton is accepted by a *disjunctive alternating parity tree automaton* [Kobayashi and Ong 2011] and that model checking of order-k recursion schemes for disjunctive alternating parity tree automata is $(k-1)$-EXPTIME. The lower bound comes from $(k-1)$-EXPTIME hardness of the reachability problem [Kobayashi and Ong 2011]. Please consult [Kobayashi and Ong 2011] about the complexity of the recursion scheme model checking problem for various subclasses of alternating parity tree automata and modal μ-calculus.

Remark 2.2. Note that we allow $\bot$ to occur in $[\![\mathcal{G}]\!]$. In particular, a recursion scheme that generates the tree consisting of a single node $\bot$ is always accepted by $\mathcal{B}^{\bot}$. This is not a problem for the purpose of verification of safety properties.

In the standard approach [Ong 2006], it is assumed that $[\![\mathcal{G}]\!]$ does not contain $\bot$. This does not lose generality in the case of the modal μ-calculus model checking, because, given a recursion scheme $\mathcal{G}$ and a modal μ-calculus formula φ, one can always construct $\mathcal{G}'$ and φ' such that (i) $[\![\mathcal{G}]\!]$ satisfies φ if and only if $[\![\mathcal{G}']\!]$ satisfies φ' (ii) $[\![\mathcal{G}']\!]$ does not contain $\bot$.

Remark 2.3. Recursion scheme model checking introduced above may look quite different from conventional, finite state model checking. The former is however considered a generalization of the latter in the following sense. In finite state model checking [Clarke et al. 1999], a finite state system is formalized as a Kripke structure $M = (\mathit{States}, I, R, L)$ where (i) States is a finite set of states, (ii) $I \subseteq \mathit{States}$ is the set of initial states, (iii) $R \subseteq \mathit{States} \times \mathit{States}$ is a transition relation that must be total, and (iv) $L \in \mathit{States} \to 2^{AP}$ is a function that maps a state to the set of atomic propositions true in the state. We can assume without loss of generality that I is a singleton set $\{s_0\}$. Define a recursion scheme $\mathcal{G}_M = (\Sigma, \mathcal{N}, \mathcal{R}, F_{s_0})$ by:

$$\Sigma = \{\mathtt{a}_P \mapsto w \mid P \in 2^{AP}\}$$
$$\mathcal{N} = \{F_s : \mathtt{o} \mid s \in \mathit{States}\}$$
$$\mathcal{R} = \{F_s \mapsto \mathtt{a}_P\, F_{s_1}\, \cdots\, F_{s_w} \mid L(s) = P, \{s_1, \ldots, s_w\} = \{s' \mid (s, s') \in R\}\}$$

Here, w is the maximum branching factor, i.e. $max_{s \in States}|\{s' \mid (s,s') \in R\}|$. In the definition of $\mathcal{R}$, the sequence $s_1, \ldots, s_w$ may contain duplicated states. $\mathcal{G}_M$ generates the computation tree [Clarke et al. 1999] of the Kripke structure M. Thus, finite state model checking can be reduced to model checking of order-0 recursion schemes.

Using deterministic trivial automata, we can express the following fragment of modal μ-calculus.

$$\varphi ::= p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid X \mid \Box\varphi \mid \nu X.\varphi$$

Here, p ranges over the set AP of atomic propositions. An important restriction is that we do not have duals of $\Box$ and ν. This fragment suffices to express safety properties that "bad things do not happen". Note however that for a formula of the fragment above, the size of the corresponding deterministic trivial automaton can be doubly exponential in the formula size. To avoid such (super) exponential blow-up of the automaton size, we need to use alternating tree automata, as in [Ong 2006; Kobayashi and Ong 2009].

3. FROM PROGRAM VERIFICATION TO RECURSION SCHEME MODEL CHECKING

This section shows that a variety of program verification problems can be reduced to recursion scheme model checking problems. We first show, in Sections 3.1–3.2, the reduction for Igarashi and Kobayashi's resource usage verification problem [Igarashi and Kobayashi 2005] in detail, because the resource usage verification is challenging due to primitives for dynamic creating an unbounded number of resources, and also because many other problems (such as reachability and flow analysis) can be easily reduced to it. We then informally discuss reductions from a variety of other verification problems to recursion scheme model checking problems in Section 3.3 in order to show the robustness of our approach. Together with decidability of the recursion scheme model checking, the reductions shown in this section imply that various program verification problems are decidable for the simply-typed λ-calculus with recursion and finite base types. For programs having infinite base types (such as integers), we can combine the methods presented below with techniques of predicate abstraction [Graf and Saïdi 1997] and CEGAR [Clarke et al. 2003; Ball and Rajamani 2002]: see [Kobayashi et al. 2011] for such an extension. For all the program verification problems considered in this section, we assume that a whole program is given. For compositional verification of programs, model checking problems need to be generalized, as briefly discussed in Section 8.

Remark 3.1. Throughout this section, we often assume that source programs are represented as a system of top-level function definitions:

$$\{f_1\,\widetilde{x}_1 = e_1, \ldots, f_n\,\widetilde{x}_n = e_n\},$$

where $e_1, \ldots, e_n$ have the unit type **unit** and expressions are evaluated in call-by-name. Programs of a call-by-value language can be transformed into the above form by using a call-by-value CPS transformation [Plotkin 1975; Danvy and Filinski 1992; Appel 1992], followed by λ-lifting [Johnsson 1985]. Consider terms of simply-typed

call-by-value λ-calculus with recursion:

$$M ::= c \mid x \mid \lambda x.M \mid M_1 M_2 \mid \mathbf{rec}(f, x, M) \mid \mathbf{if}\ M_1\ M_2\ M_3,$$

where c is a value of base type, $\mathbf{rec}(f, x, M)$ is a recursive function defined by $f(x) = M$, and $\mathbf{if}\ M_1\ M_2\ M_3$ evaluates M_1, and then evaluates M_2 if the value of M_1 is true and evaluates M_3 otherwise. Then, the following simple CPS transformation $(\cdot)^\dagger$ [Plotkin 1975] suffices.

$$\begin{aligned}
c^\dagger &= \lambda k.k\,c \\
x^\dagger &= \lambda x.k\,x \\
(\lambda x.M)^\dagger &= \lambda k.k(\lambda x.M^\dagger) \\
(M_1 M_2)^\dagger &= \lambda k.{M_1}^\dagger(\lambda f.{M_2}^\dagger(\lambda x.f\,x\,k)) \\
(\mathbf{rec}(f, x, M))^\dagger &= \lambda k.k(\mathbf{rec}(f, x, M^\dagger)) \\
(\mathbf{if}\ M_1\ M_2\ M_3)^\dagger &= \lambda k.{M_1}^\dagger \lambda x.(\mathbf{if}\ x\ ({M_2}^\dagger k)\ ({M_3}^\dagger k))
\end{aligned}$$

As noted by Meyer and Wand [1985], if M has type τ, then $M^\dagger$ has type $(\tau^\dagger \to \mathbf{unit}) \to \mathbf{unit}$, where $\tau^\dagger$ is given by:

$$b^\dagger = b \text{ (if } b \text{ is a base type)} \qquad (\tau_1 \to \tau_2)^\dagger = {\tau_1}^\dagger \to ({\tau_2}^\dagger \to \mathbf{unit}) \to \mathbf{unit}.$$

3.1 Resource Usage Verification

The goal of resource usage verification [Igarashi and Kobayashi 2005] is to statically check that a given program accesses resources (which model stateful objects like files, memory cells, and locks) in a valid manner. For example, we wish to guarantee that a resource is initialized before being accessed, that an opened file should be eventually closed and not accessed afterwards, and that an acquired lock should be eventually released.

Igarashi and Kobayashi [2005] formalized this problem for a simply-typed call-by-value λ-calculus with recursion and primitives for dynamically creating and accessing resources. Here we re-formalize it for a call-by-name λ-calculus, as it has a more direct correspondence to recursion schemes. Note that call-by-value programs can be transformed into call-by-name programs by using the call-by-value CPS transformation, as discussed in Remark 3.1.

We first introduce resource automata to specify how each resource should be used.

Definition 3.1 *(resource automaton).* A resource automaton W is a finite-state deterministic (word) automaton (L, Q, δ, q_0, Q_F), where L is a set of names of resource access primitives, Q is a set of states, δ is a partial function from $Q \times L$ to Q, $q_0 \in Q$ is an initial state, and $Q_F (\subseteq Q)$ is a set of final states.

Intuitively, $\delta(q, a) = q'$ means that a resource of state q goes to state q' after it is accessed by the primitive a. $\delta(q, a)$ is undefined if the operation a is disallowed in state q. When a program terminates, all the resources must be in final states. We fix a resource automaton $W = (L, Q, \delta, q_0, Q_F)$ below.

Definition 3.2. A *program* D is a set of *function definitions* $\{F_1\, x_{1,1} \cdots x_{1,\ell_1} = e_1, \ldots, F_n\, x_{n,1} \cdots x_{n,\ell_n} = e_n\}$, where F_i denotes a defined function symbol, and e

ranges over the set of expressions, defined by:

$$e ::= \diamond \mid x \mid F \mid e_1 e_2 \mid \mathbf{if}\_\ e_1\ e_2 \mid \mathbf{new}^q\ e \mid \mathbf{acc}_a\ x\ e$$

Here, q and a range over Q and L (of the resource automaton), respectively. We require that the function names $F_1, \ldots, F_n$ are different from each other, and that any program D contains exactly one definition for S (which is the "main" function), of the form $S = e$. Without loss of generality, we assume $S = F_1$.

The expression $\diamond$ 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 $\mathbf{new}^q\ e$ creates a resource with initial state q, and passes it to e (which is a function that takes a resource as an argument). The expression $\mathbf{acc}_a\ x\ e$ applies an operation of name a to the resource x, and then evaluates e. The name a expresses resource access primitives like `read`, `write`, and `close`.

EXAMPLE 3.1. Recall the example given in Section 1.2:

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

It can be transformed into the following program D_0 by call-by-value CPS transformation (with some optimization).

$$\begin{aligned} S &= \mathbf{new}^{q_{\texttt{read\_only}}}\ (G \diamond) \\ G\,k\,x &= \mathbf{if}\_\ (\mathbf{acc}_{\texttt{close}}\ x\ k)\ (\mathbf{acc}_{\texttt{read}}\ x\ (G\,k\,x)) \end{aligned}$$

Here, the argument k of G is a continuation parameter.

The resource automaton W is given by:

$$W = (\{\texttt{close}, \texttt{read}\}, \{q_{\texttt{read\_only}}, q_{\texttt{closed}}\}, \delta, q_{\texttt{read\_only}}, \{q_{\texttt{closed}}\})$$

where

$$\delta(q_{\texttt{read\_only}}, \texttt{read}) = q_{\texttt{read\_only}} \qquad \delta(q_{\texttt{read\_only}}, \texttt{close}) = q_{\texttt{closed}}$$

□

We consider only "well-typed" programs below.

Definition 3.3. 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 $\diamond$.

A type environment, denoted by Γ, is a map 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:

$$\Gamma \vdash \diamond : \mathbf{unit} \qquad \frac{\Gamma \vdash e : \mathbf{R} \to \mathbf{unit}}{\Gamma \vdash \mathbf{new}^q\ e : \mathbf{unit}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \Gamma\{x : \tau\} \vdash x : \tau$$

$$\frac{\Gamma \vdash e_1 : \mathbf{unit} \qquad \Gamma \vdash e_2 : \mathbf{unit}}{\Gamma \vdash \mathbf{if}\_\ e_1\ e_2 : \mathbf{unit}} \qquad \frac{\Gamma \vdash e_1 : \mathbf{R} \qquad \Gamma \vdash e_2 : \mathbf{unit}}{\Gamma \vdash \mathbf{acc}_a\ e_1\ e_2 : \mathbf{unit}}$$

A program $D = \{F_1\ x_{1,1}\ \cdots\ x_{1,k_1} = e_1, \ldots, F_n\ x_{n,1}\ \cdots\ x_{n,k_n} = e_n\}$ is *well-typed* under Γ, if $\Gamma = F_1 : \tau_{1,1} \to \cdots \to \tau_{1,k_1} \to \mathbf{unit}, \ldots, F_n : \tau_{n,1} \to \cdots \tau_{n,k_n} \to \mathbf{unit}$ and $\Gamma \cup \{x_{i,1} : \tau_{i,1}, \ldots, x_{i,k_i} : \tau_{i,k_i}\} \vdash e_i : \mathbf{unit}$ holds for each i. Here, k_i may be 0.

By the definition above, all the defined function symbols must have types of the form $\tau_1 \to \cdots \to \tau_k \to \mathbf{unit}$. That condition can be enforced by the CPS transformation, as mentioned in Remark 3.1.

We now define an operational semantics.

Definition 3.4. A *run-time state* is expressed by a pair (H, e), where e is the current expression, and H maps each resource to its state (i.e. a state of the resource automaton W). We write $[e_1/x_1, \ldots, e_\ell/x_\ell]e$ for the expression obtained by replacing every occurrence of x_i with e_i in e. The relation $(H, e) \longrightarrow_{D,W} C$, where C is a run-time state or an error configuration **Error** (which means that a resource access error has occurred), is the least relation closed under the following rules.

$$(H, F\ e_1\ \cdots\ e_\ell) \longrightarrow_{D \cup \{F\ x_1\ \cdots\ x_\ell = e\}, W} (H, [e_1/x_1, \ldots, e_\ell/x_\ell]e)$$

$$(H, \mathbf{if}\_\ e_1\ e_2) \longrightarrow_{D,W} (H, e_1)$$

$$(H, \mathbf{if}\_\ e_1\ e_2) \longrightarrow_{D,W} (H, e_2)$$

$$(H, \mathbf{new}^q\ e) \longrightarrow_{D,W} (H\{x \mapsto q\}, e\ x) \qquad (x \notin dom(H))$$

$$(H\{x \mapsto q\}, \mathbf{acc}_a\ x\ e) \longrightarrow_{D,W} (H\{x \mapsto q'\}, e) \quad (\text{if } q' = \delta(q, a))$$

$$(H\{x \mapsto q\}, \mathbf{acc}_a\ x\ e) \longrightarrow_{D,W} \mathbf{Error} \quad (\text{if } \delta(q, a) \text{ is undefined})$$

$$(H, \diamond) \longrightarrow_{D,W} \mathbf{Error} \qquad (\text{if } H(x) \notin Q_F \text{ for some } x \in dom(H))$$

Here, δ and Q_F are the transition function and the set of final states of W, respectively.

The last two rules ensure that a program is reduced to **Error** if a resource is used in an invalid manner; the first of them treats the case where an invalid access occurs, while the second one treats the case where a program has terminated but a resource is not in a final state.

Example 3.2. Recall the program D_0 in Example 3.1. It can be reduced as follows.

$$\begin{aligned}
(\emptyset, S) &\longrightarrow_{D_0,W} (\emptyset, \mathbf{new}^{q_{\mathtt{read_only}}}\ (G \diamond)) \\
&\longrightarrow_{D_0,W} (\{x \mapsto q_{\mathtt{read_only}}\}, G \diamond x) \\
&\longrightarrow_{D_0,W} (\{x \mapsto q_{\mathtt{read_only}}\}, \mathbf{if}_\ (\mathbf{acc}_{\mathtt{close}}\ x \diamond)\ (\mathbf{acc}_{\mathtt{read}}\ x\ (G \diamond x))) \\
&\longrightarrow_{D_0,W} (\{x \mapsto q_{\mathtt{read_only}}\}, \mathbf{acc}_{\mathtt{read}}\ x\ (G \diamond x)) \\
&\longrightarrow_{D_0,W} (\{x \mapsto q_{\mathtt{read_only}}\}, G \diamond x) \\
&\longrightarrow_{D_0,W} (\{x \mapsto q_{\mathtt{read_only}}\}, \mathbf{if}_\ (\mathbf{acc}_{\mathtt{close}}\ x \diamond)\ (\mathbf{acc}_{\mathtt{read}}\ x\ (G \diamond x))) \\
&\longrightarrow_{D_0,W} (\{x \mapsto q_{\mathtt{read_only}}\}, \mathbf{acc}_{\mathtt{close}}\ x \diamond) \\
&\longrightarrow_{D_0,W} (\{x \mapsto q_{\mathtt{closed}}\}, \diamond)
\end{aligned}$$

Let D_1 be the program obtained from D_0 by replacing the definition of G with:

$$G\,k\,x = \mathbf{if\_}\ (\mathbf{acc}_{\mathtt{read}}\ x\ k)\ (\mathbf{acc}_{\mathtt{close}}\ x\ (G\,k\,x))$$

Then, D_1 can be reduced as follows.

$$\begin{array}{ll}
(\emptyset, S) & \longrightarrow_{D_1,W}\ (\emptyset, \mathbf{new}^{q_{\mathtt{read_only}}}\ (G\,\diamond)) \\
& \longrightarrow_{D_1,W}\ (\{x \mapsto q_{\mathtt{read_only}}\}, G\ \diamond\ x) \\
& \longrightarrow_{D_1,W}\ (\{x \mapsto q_{\mathtt{read_only}}\}, \mathbf{if_}\ (\mathbf{acc}_{\mathtt{read}}\ x\ \diamond)\ (\mathbf{acc}_{\mathtt{close}}\ x\ (G\ \diamond\ x))) \\
& \longrightarrow_{D_1,W}\ (\{x \mapsto q_{\mathtt{read_only}}\}, \mathbf{acc}_{\mathtt{close}}\ x\ (G\ \diamond\ x)) \\
& \longrightarrow_{D_1,W}\ (\{x \mapsto q_{\mathtt{closed}}\}, G\ \diamond\ x) \\
& \longrightarrow_{D_1,W}\ (\{x \mapsto q_{\mathtt{closed}}\}, \mathbf{if_}\ (\mathbf{acc}_{\mathtt{read}}\ x\ \diamond)\ (\mathbf{acc}_{\mathtt{close}}\ x\ (G\ \diamond\ x))) \\
& \longrightarrow_{D_1,W}\ (\{x \mapsto q_{\mathtt{closed}}\}, \mathbf{acc}_{\mathtt{close}}\ x\ (G\ \diamond\ x)) \\
& \longrightarrow_{D_1,W}\ \mathbf{Error}
\end{array}$$

□

Remark 3.2. Note that the reduction is allowed only at the top-level; For example, $(H, F(\mathbf{if\_}\ e_1\ e_2)) \longrightarrow_{D,W} (H, F\ e_1)$ is not allowed. This is not a limitation, since a redex can occur only at the top-level in the result of CPS transformation.□

The following is the standard type soundness property, which means that the evaluation does not get stuck. (Note that $(\emptyset, S)$ may be reduced to **Error**, i.e. a resource access error may occur.)

LEMMA 3.1. *Let D be a well-typed program and W a resource automaton. If $(\emptyset, S) \longrightarrow^*_{D,W} (H, e)$, then either e is $\diamond$ or $(H, e) \longrightarrow_{D,W} C$ for some C.*

PROOF. This follows by the standard argument, using the facts that the reduction preserves typing, and that a well-typed run-time state does not get stuck immediately. □

We now define the problem of resource usage verification.

Definition 3.5 *(resource safety, resource usage verification).*
A (well-typed) program D is *resource-safe* (with respect to the resource automaton W) if $(\emptyset, S) \not\longrightarrow^*_{D,W}$ **Error**. The resource usage verification is the problem of deciding whether a given (well-typed) program is resource-safe or not.

EXAMPLE 3.3. The program D_0 in Example 3.1 is resource-safe. The program D_1 in Example 3.2 is not resource-safe. The following program, obtained from D_0 by removing $\mathbf{acc}_{\mathtt{close}}$, is not resource-safe either.

$$S = \mathbf{new}^{q_{\mathtt{read\_only}}}\ (G\diamond) \qquad G\,k\,x = \mathbf{if\_}\ k\ (\mathbf{acc}_{\mathtt{read}}\ x\ (G\,k\,x))$$

In fact, $(\emptyset, S)$ can be reduced to $(\{x \mapsto q_{\mathtt{read\_only}}\}, \diamond)$, but $q_{\mathtt{read\_only}}$ is not a final state. □

3.2 From Resource Usage Verification to Recursion Scheme Model Checking

The idea of reducing a resource usage verification problem to a model checking problem for recursion schemes is to transform a program into a recursion scheme that generates a tree describing all the possible resource access sequences.

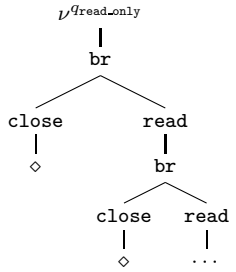


Fig. 4. A tree representing all the access sequences of the program in Example 3.1

The transformation is straightforward for a program manipulating a single resource. For example, recall the program in Example 3.1:

$$\begin{aligned} S &= \mathbf{new}^{q_{\mathtt{read\_only}}}\ (G\,\diamond) \\ G\,k\,x &= \mathbf{if}\_\ (\mathbf{acc}_{\mathtt{close}}\ x\ k)\ (\mathbf{acc}_{\mathtt{read}}\ x\ (G\,k\,x)) \end{aligned}$$

The required tree is generated by the following recursion scheme:

$$\begin{aligned} S &\to \mathbf{new}^{q_{\mathtt{read\_only}}}\ (G\,\diamond) \\ G\,k\,x &\to \mathbf{if}\_\ (\mathbf{acc}_{\mathtt{close}}\ x\ k)\ (\mathbf{acc}_{\mathtt{read}}\ x\ (G\,k\,x)) \\ \mathbf{new}^{q_{\mathtt{read\_only}}}\ k &\to \nu^{q_{\mathtt{read\_only}}}(k\,\diamond) \\ \mathbf{if}\_\ x\ y &\to \mathtt{br}\,x\,y \\ \mathbf{acc}_{\mathtt{read}}\ x\ k &\to \mathtt{read}\,k \\ \mathbf{acc}_{\mathtt{close}}\ x\ k &\to \mathtt{close}\,k \end{aligned}$$

The first two rules are identical to the source program. We have just added rules for resource access primitives and conditionals. The recursion scheme generates the tree shown in Figure 4. Thus, the resource usage verification problem has been reduced to the problem of checking whether every path of the tree generated by the recursion scheme is labeled by an element of $\nu^{q_{\mathtt{read\_only}}}(\mathtt{read}^*\mathtt{close}\,\diamond + \mathtt{read}^\omega)$ (with `br` ignored).

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

```
let rec f x y =
  if b then close(x);close(y) else (read(x);write(y);f x y) in
let z1 = open_in "foo" in let z2 = open_out "bar" in
  f z1 z2
```

It is represented as the following program in our language.

$$\begin{aligned} S &= \mathbf{new}^{q_{\mathtt{read\_only}}}\ G \\ G\ z_1 &= \mathbf{new}^{q_{\mathtt{write\_only}}}\ (F\ \diamond\ z_1) \\ F\ k\ x\ y &= \mathbf{if}\_\ (\mathbf{acc}_{\mathtt{close}}\ x\ (\mathbf{acc}_{\mathtt{close}}\ y\ k))\ (\mathbf{acc}_{\mathtt{read}}\ x\ (\mathbf{acc}_{\mathtt{write}}\ y\ (F\ k\ x\ y))) \end{aligned}$$

Since we need to verify that each of the two resources will be accessed in a valid manner, we transform the program into a recursion scheme that generates a tree that represents *resource-wise* access sequences.

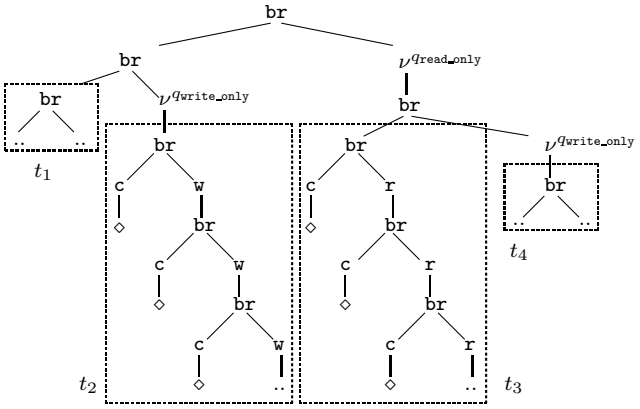


Fig. 5. A tree that represents access sequences for two resources

The recursion scheme that generates such a tree is given by:

$$
\begin{array}{rcl}
S & \to & \mathbf{new}^{q_{\mathtt{read\_only}}}\ G \\
G\,z_1 & \to & \mathbf{new}^{q_{\mathtt{write\_only}}}\ (F \diamond z_1) \\
F\,k\,x\,y & \to & \mathbf{if}_{\_}\ (\mathbf{acc}_{\mathtt{close}}\ x\ (\mathbf{acc}_{\mathtt{close}}\ y\ k))\ (\mathbf{acc}_{\mathtt{read}}\ x\ (\mathbf{acc}_{\mathtt{write}}\ y\ (F\,k\,x\,y))) \\
\mathbf{new}^{q_{\mathtt{read\_only}}}\ k & \to & \mathtt{br}\,(k\,K)\,(\nu^{q_{\mathtt{read\_only}}}(k\,I)) \\
\mathbf{new}^{q_{\mathtt{write\_only}}}\ k & \to & \mathtt{br}\,(k\,K)\,(\nu^{q_{\mathtt{write\_only}}}(k\,I))
\end{array}
$$

$$
\mathbf{acc}_{\mathtt{read}}\ x\ k \to x\,\mathtt{read}\,k \quad \mathbf{acc}_{\mathtt{write}}\ x\ k \to x\,\mathtt{write}\,k \quad \mathbf{acc}_{\mathtt{close}}\ x\ k \to x\,\mathtt{close}\,k
$$

$$
I\ f\ k \to f(k) \qquad K\ f\ k \to k \qquad \mathbf{if}_{\_}\ x\ y \to \mathtt{br}\,x\,y
$$

Again, the first three rules are the same as the function definitions in the source program. The trick to handle multiple resources is in the rule for $\mathbf{new}^q$, where the resource is non-deterministically instantiated to I or K, of sort $(\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$.[4] I and K take a resource access operation f as an argument; I attaches f to the access tree, while K just ignores f. 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. A resource access $\mathbf{acc}_a\ x\ k$ is now transformed into $x\ a\ k$, which is reduced to $a(k)$ or k, depending on whether x is I or K.

Figure 5 shows the tree generated by the above recursion scheme. The four subtrees marked by dashed boxes express possible access sequences obtained by keeping track of different resources. In t_1, both the files are ignored. In t_2, the write-only file is tracked, while in t_3, the read-only file is tracked. Both the files are tracked in t_4. For the purpose of resource usage verification, we need to check only t_2 and t_3; The subtrees t_1 and t_4, which either contain no ν or more than one ν, can be ignored.

We now formally define a transformation from a resource usage verification problem to a model checking problem for recursion schemes, i.e., a pair $(\mathcal{G}, \mathcal{B})$ such that the source program is resource-safe if and only if the tree generated by $\mathcal{G}$ is accepted by $\mathcal{B}$.

[4]This trick was inspired from a technique used in finite state model checking [Cook et al. 2007].

We first give a transformation of a program into a recursion scheme.

Definition 3.6. Let D be a program well-typed under Γ and $W = (L, Q, \delta, q_0, Q_F)$ a resource automaton. The recursion scheme $\mathcal{G}_{D,W}$ is $(\Sigma, \mathcal{N}, \mathcal{R}, S)$, where:

$$\begin{array}{l}
\Sigma = \{a \mapsto 1 \mid a \in L\} \cup \{\mathtt{br} \mapsto 2, \diamond \mapsto 0\} \cup \{\nu^q \mapsto 1 \mid q \in Q\} \\
\mathcal{N} = \{F : (\Gamma(F))^\sharp \mid F \in dom(\Gamma)\} \\
\quad \cup \{\mathbf{acc}_a : ((\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o} \mid a \in L\} \\
\quad \cup \{\mathbf{new}^q : (((\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}) \to \mathtt{o}) \to \mathtt{o} \mid q \in Q\} \\
\quad \cup \{\mathbf{if}_ : \mathtt{o} \to \mathtt{o} \to \mathtt{o}, I : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}, K : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}\} \\
\mathcal{R} = \{F \mapsto \lambda\widetilde{x}.e \mid F\,\widetilde{x} = e\} \\
\quad \cup \{\mathbf{new}^q \mapsto \lambda k.(\mathtt{br}\,(k\,K)\,(\nu^q\,(k\,I))) \mid q \in Q\} \\
\quad \cup \{\mathbf{if}_ \mapsto \lambda x.\lambda y.(\mathtt{br}\,x\,y),\ I \mapsto \lambda f.\lambda k.f(k),\ K \mapsto \lambda f.\lambda k.k\} \\
\quad \cup \{\mathbf{acc}_a \mapsto \lambda x.\lambda k.(x\,a\,k) \mid a \in L\}
\end{array}$$

Here, $(\cdot)^\sharp$ is a translation from types (of our programming language) to sorts, given by:

$$\mathbf{R}^\sharp = (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o} \qquad \mathbf{unit}^\sharp = \mathtt{o} \qquad (\tau_1 \to \tau_2)^\sharp = \tau_1^\sharp \to \tau_2^\sharp$$

The following lemma guarantees that $\mathcal{G}_{D,W}$ is indeed a higher-order recursion scheme.

Lemma 3.2. *If W is a resource automaton and D is a well-typed program, then $\mathcal{G}_{D,W}$ is a higher-order recursion scheme.*

Proof. It suffices to check that all the rewriting rules are well-sorted. It is easy to see that if $\Gamma \vdash e : \tau$, then e is a term of sort $\tau^\sharp$ under the sorting context $\Gamma^\sharp$ (where $(x_1{:}\tau_1, \ldots, x_n{:}\tau_n)^\sharp = x_1{:}\tau_1^\sharp, \ldots, x_n{:}\tau_n^\sharp$), which implies that all the rewriting rules are well-sorted. □

Next we give a transformation of a resource automaton to a trivial automaton for recognizing trees representing valid event sequences.

Definition 3.7. Let $W = (L, Q, \delta, q_0, Q_F)$ be a resource automaton. The trivial automaton $\mathcal{B}_W$ is $(\Sigma, Q', \delta', q_{\mathbf{untracked}})$, where:

$$\begin{array}{l}
\Sigma = \{a \mapsto 1 \mid a \in L\} \cup \{\mathtt{br} \mapsto 2, \diamond \mapsto 0\} \cup \{\nu^q \mapsto 1 \mid q \in Q\} \\
Q' = Q \cup \{q_{\mathbf{untracked}}, q_{\mathbf{any}}\} \text{ (where } q_{\mathbf{untracked}}, q_{\mathbf{any}} \notin Q) \\
\delta' = \delta \cup \{(q, \mathtt{br}) \mapsto q\,q \mid q \in Q'\} \\
\quad \cup \{(q, \diamond) \mapsto \epsilon \mid q \in Q_F \cup \{q_{\mathbf{untracked}}, q_{\mathbf{any}}\}\} \\
\quad \cup \{(q_{\mathbf{untracked}}, \nu^q) \mapsto q \mid q \in Q\} \\
\quad \cup \{(q, \nu^{q'}) \mapsto q_{\mathbf{any}} \mid q, q' \in Q\} \\
\quad \cup \{(q_{\mathbf{any}}, a) \mapsto q_{\mathbf{any}} \mid a \in L\}
\end{array}$$

The 3rd set of δ describes transitions for the program termination. The transitions are defined if the tracked resource (i.e., the resource that has been instantiated to I) has been used up (i.e. if $q \in Q_F$), if no resource has been instantiated to I (i.e., if $q = q_{\mathbf{untracked}}$), or if more than one resource has been instantiated to I (i.e., if $q = q_{\mathbf{any}}$). The 4th set of δ describes transitions for the case where a tracked resource has been created for the first time, while the 5th set describes transitions

for the case where more than one tracked resource has been created. In the latter case, all the following accesses are ignored by the last set of transition rules of δ.

EXAMPLE 3.4. Let W be a resource automaton:

$$(\{\mathtt{read}, \mathtt{write}, \mathtt{close}\}, Q, \delta, q_{\mathtt{read\_only}}, \{q_{\mathtt{closed}}\}),$$

where $Q = \{q_{\mathtt{read\_only}}, q_{\mathtt{write\_only}}, q_{\mathtt{closed}}\}$ and

$$\begin{array}{ll} \delta(q_{\mathtt{read\_only}}, \mathtt{read}) = q_{\mathtt{read\_only}} & \delta(q_{\mathtt{read\_only}}, \mathtt{close}) = q_{\mathtt{closed}} \\ \delta(q_{\mathtt{write\_only}}, \mathtt{write}) = q_{\mathtt{write\_only}} & \delta(q_{\mathtt{write\_only}}, \mathtt{close}) = q_{\mathtt{closed}} \end{array}$$

$\mathcal{B}_W$ has states: $Q' = Q \cup \{q_{\mathbf{untracked}}, q_{\mathbf{any}}\}$. The transition function δ' of $\mathcal{B}_W$ has the following rules, in addition to the same rules as δ above.

$\delta'(q, \mathtt{br}) = q\,q$ (for each $q \in Q'$)
$\delta'(q_{\mathbf{untracked}}, \diamond) = \delta'(q_{\mathbf{any}}, \diamond) = \delta'(q_{\mathtt{closed}}, \diamond) = \epsilon$
$\delta'(q_{\mathbf{untracked}}, \nu^q) = q$ (for each $q \in Q$) $\quad \delta'(q, \nu^{q'}) = q_{\mathbf{any}}$ (for each $q, q' \in Q$)
$\delta'(q_{\mathbf{any}}, a) = q_{\mathbf{any}}$ (for each $a \in \{\mathtt{read}, \mathtt{write}, \mathtt{close}\}$)

□

The correctness of the transformation is stated as follows.

THEOREM 3.3. *Let D be a (well-typed) program and W a resource automaton (for D). Then, D is resource-safe with respect to W if and only if $[\![\mathcal{G}_{D,W}]\!]$ is accepted by ${\mathcal{B}_W}^{\perp}$.*

Note that the theorem above holds for programs that create an arbitrary number of resources. To prove the theorem, we just need to relate a reduction sequence of the program D to a run of the automaton $\mathcal{B}_W$ over a path of the tree generated by $\mathcal{G}_{D,W}$. Since the proof of the above theorem is tedious but not difficult, we defer the proof to Appendix A.

The following is an immediate corollary of Corollary 2.2 and Theorem 3.3.

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

Remark 3.3. We have so far considered programs having only the unit value $\diamond$ as a base value. Other finite base types such as booleans can be encoded by using the standard Church encoding. An element v_i of a finite base type consisting of $\{v_1, \ldots, v_n\}$ can be represented by the function:

$$\lambda x_1. \cdots \lambda x_n. x_i : \underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{n} \to \mathtt{o}.$$

A case expression: $\mathtt{case}\ x\ \mathtt{of}\ v_1 \Rightarrow e_1 \mid \cdots \mid v_n \Rightarrow e_n$ can be encoded into $x\ e_1\ \cdots\ e_n$. Here, in the case expressions above, we assume that $e_1, \ldots, e_n$ have type **unit** (which can be enforced by CPS transformation), so that they have type $\mathtt{o}$ after the encoding. Thus, the decidability of resource usage verification remains valid for the language extended with finite base types.

Remark 3.4. Instead of generating a single recursion scheme and a trivial automaton from a program, we can also generate a pair of a recursion scheme and a

trivial automaton *for each occurrence of* $\mathbf{new}^q$. Then, the resource usage verification problem is reduced to *a set of* model checking problems for recursion schemes. For instance, recall the program accessing two resources:

$$S = \mathbf{new}^{q_{\mathtt{read\_only}}}\ G$$
$$G\, z_1 = \mathbf{new}^{q_{\mathtt{write\_only}}}\ (F \diamond z_1)$$
$$F\, k\, x_1\, x_2 = \mathbf{if}\_\ (\mathbf{acc}_{\mathtt{close}}\ x_1\ (\mathbf{acc}_{\mathtt{close}}\ x_2\ k))\ (\mathbf{acc}_{\mathtt{read}}\ x_1\ (\mathbf{acc}_{\mathtt{write}}\ x_2\ (F\, k\, x_1\, x_2)))$$

It can be transformed into the following two recursion schemes

$$\begin{aligned}
\mathcal{R}_1 \;=\; \{&S \to \mathbf{new}^{q_{\mathtt{read_only}}}\ G_1, \quad G\ z_1 \to F \diamond z_1\, K,\\
&F\ k\ x_1\ x_2 \to \mathbf{if}_\ (\mathbf{acc}_{\mathtt{close}}\ x_1\ (\mathbf{acc}_{\mathtt{close}}\ x_2\ k))\\
&\qquad\qquad (\mathbf{acc}_{\mathtt{read}}\ x_1\ (\mathbf{acc}_{\mathtt{write}}\ x_2\ (F\, k\, x_1\, x_2))), \ldots\}\\
\mathcal{R}_2 \;=\; \{&S \to G_2\ K, \quad G_2\ z_1 \to \mathbf{new}^{q_{\mathtt{write_only}}}\ (F \diamond z_1)),\\
&F\ k\ x_1\ x_2 \to \mathbf{if}_\ (\mathbf{acc}_{\mathtt{close}}\ x_1\ (\mathbf{acc}_{\mathtt{close}}\ x_2\ k))\\
&\qquad\qquad (\mathbf{acc}_{\mathtt{read}}\ x_1\ (\mathbf{acc}_{\mathtt{write}}\ x_2\ (F\, k\, x_1\, x_2))), \ldots\}
\end{aligned}$$

This alternative approach may be preferable in practice, as the size of the corresponding trivial automaton is kept small. □

Remark 3.5. We can extend the encoding to verify some inter-dependency between the uses of different resources. Suppose that resources should be used in a LIFO manner, in the sense that the most recently created resource should be closed first. To verify that property, we can change the encoding of resource primitives as follows:

$$\mathbf{new}^q\, k \to \mathtt{br}\,(k\, K)\,(\mathtt{br}\,(\nu_1^q\,(k\, I_1))\,(\nu_2^q\,(k\, I_2)))$$
$$I_1\, x\ k \to \mathtt{first}(x(k)) \qquad I_2\, x\ k \to \mathtt{second}(x(k))$$

Then, it suffices to check that, for every path of the form $s_0\nu_1^q s_1 \nu_2^q s_2$ where $s_0 s_1 s_2$ does not contain ν_i^q, a subsequence $\mathtt{first} \cdot \mathtt{close}$ occurs only after $\mathtt{second} \cdot \mathtt{close}$. This property can be expressed by a trivial automaton.

3.3 Other Verification Problems

This subsection discusses reductions of other verification problems to model checking problems for recursion schemes, in order to show the robustness of our approach to program verification. The discussion is brief and rather informal; The formalization and correctness proofs of the reductions would be tedious but not difficult.

3.3.1 *Reachability.* The language we consider here is a sub-language of the one considered in Section 3.1, obtained by removing resource primitives $\mathbf{new}^q\ e$ and $\mathbf{acc}_a\ x\ e$. The operational semantics for this sub-language is obtained by just removing the "H" component from run-time states. The evaluation is thus call-by-name.

The *reachability* is the problem of checking whether the main function S is reduced to *Fail*, which is a special function symbol of type $\mathtt{o}$.

Given a program D well-typed under Γ, construct the following recursion scheme $\mathcal{G}_D$ and automaton $\mathcal{B}$:

$$\mathcal{G}_D = (\{\mathtt{fail} \mapsto 0, \diamond \mapsto 0, \mathtt{br} \mapsto 2\}, \Gamma^\sharp, \mathcal{R}, S)$$
$$\mathcal{R} = \{F\ \widetilde{x} \to e \mid F\ \widetilde{x} = e \in D, \text{ and } F \neq \mathit{Fail}\} \cup \{\mathit{Fail} \to \mathtt{fail}, \mathbf{if}\_\ x\ y \to \mathtt{br}\, x\, y\}$$
$$\mathcal{B} = (\{\mathtt{fail} \mapsto 0, \diamond \mapsto 0, \mathtt{br} \mapsto 2\}, \{q_0\}, \{(q_0, \mathtt{br}) \mapsto q_0 q_0, (q_0, \diamond) \mapsto \epsilon\}, q_0)$$

It is trivial that S is reduced to *Fail* if and only if $[\![\mathcal{G}_D]\!]$ contains `fail`. Thus, S is reduced to *Fail* if and only if $[\![\mathcal{G}_D]\!]$ is not accepted by $\mathcal{B}^{\perp}$. The reachability problem is therefore decidable.

EXAMPLE 3.5. Consider the following program in a call-by-value language.

```
let repeatEven f x = if _ then x else f (repeatOdd f x) in
let repeatOdd f x = f (repeatEven f x) in
  if (repeat_even not true) then () else fail
```

The function `repeatEven` takes two arguments f and x, and applies f to x an even number of times.

It can be transformed into the following program in our language.

$$\begin{array}{l} S = \mathit{RepeatEven}\ C\ \mathit{Not}\ \mathit{True} \qquad C\ b = \mathit{If}\ b\ \diamond\ \mathit{Fail} \\ \mathit{RepeatEven}\ k\ f\ x = \mathbf{if\_}\ (k\ x)\ (\mathit{RepeatOdd}\ (f\ k)\ f\ x) \\ \mathit{RepeatOdd}\ k\ f\ x = \mathit{RepeatEven}\ (f\ k)\ f\ x \\ \mathit{Not}\ k\ b = \mathit{If}\ b\ (k\ \mathit{False})\ (k\ \mathit{True}) \\ \mathit{If}\ b\ x\ y = b\ x\ y \qquad \mathit{True}\ x\ y = x \qquad \mathit{False}\ x\ y = y \end{array}$$

Here, we have applied CPS transformation and used Church encoding for booleans. The corresponding recursion scheme can be obtained by just adding the rules for *Fail* and $\mathbf{if\_}$, as described above. The resulting recursion scheme generates a tree containing `fail` if and only if the original program reaches *Fail*. We can use recursion scheme model checking to show that the former is not the case, which implies that the original program does not reach *Fail*. □

3.3.2 *Call Relation Analysis and Control Flow Analysis.* We consider the same language as above. Given a program $D = \{F_1\ \widetilde{x} = e_1, \ldots, F_n\ \widetilde{x} = e_n\}$ (with the main function S), we define the call relation $\mathbf{Call}_D$ by:

$$\mathbf{Call}_D = \{(F_i, F_j) \mid S \longrightarrow_D^* F_i\,\widetilde{t} \longrightarrow_D (\overset{\mathtt{if}}{\longrightarrow}_D)^* F_j\,\widetilde{u}\}.$$

Here, the relation $(\overset{\mathtt{if}}{\longrightarrow}_D)^*$ represents a (possibly empty) sequence of reductions of if-expressions. We define the *call relation analysis* as the problem of, given D, F_i, F_j, to decide whether $(F_i, F_j) \in \mathbf{Call}_D$. Note that solving the problem is not obvious syntactically, as our language is higher-order, and the head term of the body of the definition of F_i may be a variable.

The call relation analysis above can be easily reduced to recursion scheme model checking. The idea is to transform a program into a recursion scheme that generates a tree representing all the possible call sequences. For each function F_i in a program D, prepare a terminal symbol $\mathtt{call}_i$, and transform a function definition $F_i\ \widetilde{x} = e$ to

$$F_i\ \widetilde{x} \to \mathtt{call}_i(e)$$

and add the rule $\mathbf{if\_}\ x\ y \to \mathtt{br}\,x\,y$ for if-expressions. Then, it is obvious that $(F_i, F_j) \in \mathbf{Call}_D$ if and only if the tree generated by the recursion scheme has a path of the form $\cdots \mathtt{call}_i \mathtt{br}^* \mathtt{call}_j \cdots$. Since the latter is a recursion scheme model checking problem, the call relation analysis is decidable.

The call relation analysis above is strongly related to a control flow analysis problem [Nielson et al. 1999] of computing, for each function application $M_1 M_2$,

the set of (syntactic occurrences of) λ-abstractions that M_1 may evaluate to. Given a call-by-value program, apply the CPS transformation given in Remark 3.1. Recall that a function application $M_1 M_2$ is transformed to:

$$\lambda k.{M_1}^\dagger(\lambda f.{M_2}^\dagger(\lambda x.f\,x\,k)).$$

Thus, the value of M_1 will be assigned to f and called in the expression $f\,x\,k$. Let F_i be the name of the continuation function $\lambda x.f\,x\,k$. Then, the call relation for the program in CPS contains (F_i, F_j) if and only if, in the source program, the value of M_1 evaluates to a closure corresponding to F_j and is called in $M_1 M_2$.

EXAMPLE 3.6. Let us consider the following call-by-value program, taken from Might and Shivers [2008].

```
let lam = fun x->... in let lam' = fun x->... in
let id x = x in let unused = id lam in let f = id lam' in
   f()
```

Suppose we are interested in whether `f` in the call `f()` may be bound to `lam`.

By applying CPS transformation (with some simplification), we obtain the following program D in our language.

$$\begin{array}{lll} S = Id\ Lam\ C_1 & C_1\ unused = Id\ Lam'\ C_2 & C_2\,f = f \diamond \\ Id\ x\ k = k\,x & Lam\ x = \cdots & Lam'\ x = \cdots \end{array}$$

The problem has been reduced to the call relation problem of whether $(C_2, \mathtt{Lam}) \in \mathbf{Call}_D$ holds. The program above can be further transformed to the following recursion scheme:

$$\begin{array}{lll} S \to \mathtt{call}_0(Id\ Lam\ C_1) & C_1\ unused \to \mathtt{call}_0(Id\ Lam'\ C_2) & C_2\ f \to \mathtt{call}_1(f \diamond) \\ Id\ x\ k \to \mathtt{call}_0(k\,x) & Lam\ x \to \mathtt{call}_2(\cdots) & Lam'\ x \to \mathtt{call}_0(\cdots) \end{array}$$

Here we have assigned the label $\mathtt{call}_0$ to irrelevant functions. The tree generated by the recursion scheme does not have $\mathtt{call}_1 \cdot \mathtt{call}_2$ as a subpath. Thus, we know $(C_2, Lam) \notin \mathbf{Call}_D$.

□

Remark 3.6. Another way to reduce the flow analysis problem into recursion scheme model checking is to first reduce flow analysis into a resource usage verification problem. Given a functional program, we can transform every λ-abstraction into a pair consisting of the λ-abstraction and a resource. The resource is used to keep track of uses of the function. We can also transform every function application $e_1 e_2$ into an access to the resource followed by the function application (i.e. $\mathbf{let}\ (f, r) = e_1\ \mathbf{in\ let}\ y = e_2\ \mathbf{in\ acc}_a\ r\ (f\,y)$). We can then check which λ-abstraction is called at each functional call site by solving resource usage verification problems.

3.3.3 *Exception Analysis.* The goal of exception analysis [Yi 1994; Leroy and Pessaux 2000; Nielson et al. 1999] is to check whether there is an uncaught exception. If a program uses only finitely many exceptions that carry base values, we can transform out exception primitives by representing exception handlers as alternative continuations, following Blume et al. [2008], and reduce the exception analysis problem to the reachability problem, which can be further reduced to a model checking problem as discussed in Section 3.3.1.

For example, consider the following OCaml program, taken from Leroy and Pessaux [2000]:

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

By representing exception handlers as alternative continuations, we get the following program in our language.

$S = G\ \diamond\ \ \mathit{Uncaught}\ \mathit{End}$
$F\ x\ h\ k = \mathbf{if\_}\ (k\ x)\ (\mathit{Failwith}\ \mathit{True}\ h)$ $\quad G\,x\,h\,k = F\,x\,(\mathit{Handle}\,h\,k)\,k$
$\mathit{Failwith}\ x\ h = h\,x$ $\quad \mathit{Handle}\ h\ k\ x = \mathit{If}\ x\ (k\ \diamond)\ (h\ x)$
$\mathit{Uncaught}\ x = \mathit{Fail}$ $\quad \mathit{End}\ x = \mathtt{end}$

Here, *If* and *True* are defined as in Example 3.5. The functions F and G take a handler h and a continuation k as additional parameters. The program above evaluate to *Fail* if and only if the original program raises an uncaught exception. Thus, the exception analysis has been reduced to the reachability problem.

3.3.4 *Strictness Analysis.* The goal of *strictness analysis* [Mycroft 1980; Burn et al. 1986] is to check whether a function argument can be eagerly evaluated without changing the semantics of a lazy functional language. For that purpose, it suffices to check whether the function argument is used in any non-divergent execution of the program. Thus, for our language, strictness can be defined as follows.

Definition 3.8. A function F is *strict in the i-th argument* in a program D if in every finite reduction sequence of the form:

$$S \longrightarrow_D^* F\,t_1\,\cdots\,t_n \longrightarrow_D^* \diamond,$$

t_i occurs in a head position[5] in the reduction sequence $F\,t_1\,\cdots\,t_n \longrightarrow_D^* \diamond$.

Remark 3.7. Note that the above definition is slightly different from the standard definition of strictness, where a function $f(x_1, \ldots, x_n)$ is strict in x_i if $f(v_1, \cdots, v_{i-1}, \bot, v_{i+1}, \cdots, v_n) = \bot$ for *all* $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$. In our definition, the arguments $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ are restricted to actual arguments that F is applied to in reduction sequences from S.

We can reduce the strictness analysis problem above to a recursion scheme model checking problem, by replacing the definition $F\,x_1\,\cdots\,x_n = e$ with the following rules:

$F\ x_1\ \cdots\ x_n \to \mathtt{br}\ (F'\ x_1\ \cdots\ x_n)\ (\mathtt{tracked}(F'\ x_1\ \cdots\ x_{i-1}\ (\mathit{Mark}\ x_i)\ x_{i+1}\ \cdots, x_n))$
$F'\ x_1\ \cdots\ x_n \to e$
$\mathit{Mark}\ x\ \widetilde{y} \to \mathtt{used}(x\,\widetilde{y})$.

The idea is the same as that of the reduction of resource usage verification problems; We just non-deterministically focus on a function call for F, and keep track of a use

[5]More precisely, a copy of t_i that originates from the i-th argument of F occurs in a head position. We assume that each term is implicitly labeled to indicate the origin of the term.

of the i-th argument. Then, a function F is strict in the i-th argument if, and only if, in every finite path of the tree generated by the recursion scheme, `used` occurs whenever `tracked` occurs.

EXAMPLE 3.7. Let us consider the following program:

```
let f x y g = if x then g y else f _ y g in
let g y = y in f false true g
```

It is expressed in our language as follows.

$$\begin{array}{rcl} S & = & F\ \mathit{False}\ \mathit{True}\ G\ C \\ F\ x\ y\ g\ k & = & \mathit{If}\ x\ (G\ y\ k)\ (\mathbf{if\_}\ (F\ \mathit{True}\ y\ g\ k)\ (F\ \mathit{False}\ y\ g\ k)) \\ G\ y\ k & = & k\ y \\ C\ b & = & b\ \diamond\ \diamond \end{array}$$

To check whether F is strict in the second argument, it suffices to transform the program into the following recursion scheme:

$$\begin{array}{rcl} S & \to & F\ \mathit{False}\ \mathit{True}\ G\ C \\ F\ x\ y\ g\ k & \to & \mathtt{br}\ (F'\ x\,y\,g\,k)\ (\mathtt{tracked}(F'\ x\,(\mathit{Mark}\ y)\,g\,k)) \\ F'\ x\ y\ g\ k & \to & \mathit{If}\ x\ (G\ y\ k)\ (\mathtt{br}\ (F\ \mathit{True}\ y\ g\ k)\ (F\ \mathit{False}\ y\ g\ k)) \\ \mathit{Mark}\ y\ z\ w & \to & \mathtt{used}(y\ z\ w) \\ G\ y\ k & \to & k\ y \\ & \cdots & \end{array}$$

For every finite path p of the tree generated by the recursion scheme above, p contains `used` whenever p contains `tracked`. Thus, we know that F is strict in the second argument.

□

4. FROM RECURSION SCHEME MODEL CHECKING TO TYPE CHECKING

We have seen in the previous section that various verification problems for higher-order programs can be reduced to recursion scheme model checking. As the recursion scheme model checking is decidable [Ong 2006], we can, in principle, solve those program verification problems by appealing to a model checking algorithm for recursion schemes. There remains however an important question from a practical point of view: Is the resulting verification method feasible? Since the reduction to recursion scheme model checking can be efficiently performed (as we have seen, it essentially amounts to performing CPS transformation and λ-lifting), the main question is whether there is an efficient algorithm for recursion scheme model checking.

Unfortunately, the modal μ-calculus model checking of an order-k recursion scheme is k-EXPTIME complete in general. Even for the class of deterministic trivial automata considered in this article, the recursion scheme model checking is $(k-1)$-EXPTIME complete [Kobayashi and Ong 2011]. Furthermore, from the complexity of recursion scheme model checking, we can also easily deduce that the verification problems we have considered in the previous section are also $(k-1)$-EXPTIME hard [Kobayashi and Ong 2011]. Thus, from the viewpoint of complexity theory, our verification method is infeasible.

There is still a hope, however, that our verification method may work well in practice. First, k-EXPTIME hardness is the worst-case complexity: It may be the case that recursion scheme model checking problems can be efficiently solved for typical inputs. As an analogy, ML typability is DEXPTIME-complete [Mairson 1990; Kfoury et al. 1990], but the exponential behavior is rarely seen in practice. Secondly, k-EXPTIME completeness is in the combined size of a recursion scheme and a property (expressed by either a modal μ-calculus formula or a tree automaton). If the time complexity is polynomial in the size of the recursion scheme, then model checking of order-k recursion schemes may scale for simple properties and small k. Unfortunately, the previous algorithms for recursion scheme model checking [Ong 2006; Aehlig 2007; Hague et al. 2008] gave no hint about whether this is the case. Those algorithms suffer from k-EXPTIME bottleneck for almost all inputs, not just for the worst-case input.

To circumvent the situation, we introduce new, type-based model checking algorithms for recursion schemes in the present and next sections. This section shows that recursion scheme model checking can be reduced to the typability problem in an intersection type system: The tree generated by a recursion scheme satisfies a given property if, and only if, the recursion scheme is well-typed in the intersection type system. As discussed in Section 4.3, the reduction to the typability problem yields a naive fixed-point computation algorithm for model checking recursion schemes. Remarkably, despite its simplicity, the algorithm runs in time *linear* in the size of the recursion scheme, under the assumption that the sizes of sorts and properties are bounded by a constant. As the naive algorithm is still impractical due to a huge constant factor (that is k-fold exponential in the size of a property and the largest arity of non-terminals), Section 5 gives a more elaborate model checking algorithm that runs much faster for typical inputs.

4.1 Type System for Recursion Schemes

Let $\mathcal{B} = (\Sigma_{\mathcal{B}}, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_{\mathcal{B},0})$ be a trivial automaton. We omit the subscript $\mathcal{B}$ when it is clear from the context. We shall construct a type system for higher-order recursion schemes, such that a recursion scheme $\mathcal{G}$ has type $q_{\mathcal{B},0}$ if and only if $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$.

The idea is to refine the tree sort $\mathtt{o}$ to an intersection type [Coppo et al. 1979; Barendregt et al. 1983; van Bakel 1995] of the form $q_1 \wedge \cdots \wedge q_k$. Intuitively, q_i describes trees that are accepted by $\mathcal{B}$ from state q_i (i.e., accepted by $(\Sigma_{\mathcal{B}}, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_i)$). The type $q_1 \wedge \cdots \wedge q_k$ 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 instance, in Example 2.3, $\mathtt{b}$ has type $(q_1 \to q_0) \wedge (q_1 \to q_1)$. The terminal $\mathtt{br}$ has type $\bigwedge\{q \to q \to q \mid q \in \{q_0, q_1\}\}$.

We now introduce the formal syntax of types. Following van Bakel [1992], we impose the restriction that intersection type constructors occur only on the lefthand side of function type constructors. For each sort κ, we introduce two kinds of intersection types: arrow types and intersection types.

Definition 4.1 *(intersection types).* Let $\mathcal{B} = (\Sigma_{\mathcal{B}}, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_{\mathcal{B},0})$ be a trivial automaton. For each sort κ, the set $\mathbf{ATypes}_{\kappa,\mathcal{B}}$ of *arrow types* and the set $\mathbf{ITypes}_{\kappa,\mathcal{B}}$

of *intersection types* are defined as follows by induction on κ.

$$\begin{array}{rcl}\mathbf{ATypes}_{\mathsf{o},\mathcal{B}} &=& Q_{\mathcal{B}}\\ \mathbf{ATypes}_{\kappa_1\to\kappa_2,\mathcal{B}} &=& \{\sigma\to\theta \mid \sigma\in\mathbf{ITypes}_{\kappa_1,\mathcal{B}}, \theta\in\mathbf{ATypes}_{\kappa_2,\mathcal{B}}\}\\ \mathbf{ITypes}_{\kappa,\mathcal{B}} &=& \{\bigwedge S \mid S\subseteq \mathbf{ATypes}_{\kappa,\mathcal{B}}\}\end{array}$$

We use meta-variables θ and σ for arrow types and intersection types, respectively. We often write $\theta_1\wedge\cdots\wedge\theta_n$ or $\bigwedge_{i\in\{1,\ldots,n\}}\theta_i$ for the intersection type $\bigwedge\{\theta_1,\ldots,\theta_n\}$, and write $\top$ for $\bigwedge\emptyset$. Since S in the intersection type $\bigwedge S$ is a set, the order of elements and the number of occurrences of elements in S are irrelevant. Thus, for example, $\theta_1\wedge\theta_2$, $\theta_2\wedge\theta_1$, and $\theta_1\wedge\theta_1\wedge\theta_2$ are the same types.

We shall construct below a type system to reason about the type of the tree generated by a higher-order recursion scheme.

Definition 4.2 *(type environment).* A *type environment* is a set of bindings of the form $x:\theta_i$, which may contain multiple bindings for the same variable. We write $dom(\Gamma)$ for the set $\{x \mid \exists\theta. x:\theta\in\Gamma\}$. Let $\mathcal{K}$ be a map from variables to sorts. We say Γ *respects* $\mathcal{K}$, written $\Gamma ::_{\mathcal{B}} \mathcal{K}$, if (i) $dom(\Gamma)\subseteq dom(\mathcal{K})$, and (ii) $\theta\in\mathbf{ATypes}_{\mathcal{K}(x),\mathcal{B}}$ for every $x:\theta\in\Gamma$.

Note that each variable may occur more than once in a type environment; Such type environments have been used in some intersection or polymorphic type systems [Damas 1984; Coppo et al. 1981]. Intuitively, $\{x:\theta_1, x:\theta_2\}$ means that x has the intersection type $\theta_1\wedge\theta_2$. In the definition of $\Gamma ::_{\mathcal{B}}\mathcal{K}$, a variable bound in $\mathcal{K}$ but not in Γ can be considered to have type $\top$, i.e. $\bigwedge\emptyset$.

We next introduce type judgment for λ-terms. Non-terminal symbols of a recursion scheme are treated as variables below.

Definition 4.3 *(typing for terms).* Let $\mathcal{B}=(\Sigma,Q_{\mathcal{B}},\delta_{\mathcal{B}},q_{\mathcal{B},0})$ be a trivial automaton. The *type judgment relation* $\Gamma\vdash_{\mathcal{B}} t:\theta$, where Γ is a type environment, t is a λ-term, and θ is an arrow type, is the least relation closed under the following rules:

$$\Gamma\cup\{x:\theta\}\vdash_{\mathcal{B}} x:\theta \qquad \text{(T-Var)}$$

$$\frac{\delta_{\mathcal{B}}(q,a)=q_1\cdots q_n}{\Gamma\vdash_{\mathcal{B}} a: q_1\to\cdots\to q_n\to q} \qquad \text{(T-Const)}$$

$$\frac{\Gamma\vdash_{\mathcal{B}} t_1:\bigwedge_{i\in\{1,\ldots,n\}}\theta_i\to\theta \qquad \Gamma\vdash_{\mathcal{B}} t_2:\theta_i \text{ (for each } i\in\{1,\ldots,n\})}{\Gamma\vdash_{\mathcal{B}} t_1 t_2:\theta} \qquad \text{(T-App)}$$

$$\frac{\Gamma\cup\{x:\theta_1,\ldots,x:\theta_n\}\vdash_{\mathcal{B}} t:\theta \qquad x\notin dom(\Gamma)}{\Gamma\vdash_{\mathcal{B}}\lambda x.t:\bigwedge_{i\in\{1,\ldots,n\}}\theta_i\to\theta} \qquad \text{(T-Abs)}$$

Above are standard typing rules for an intersection type system, except that the type of a terminal is determined by the transition function of $\mathcal{B}$. In T-App, if t_1 has type $\bigwedge_{i\in\{1,\ldots,n\}}\theta_i\to\theta$, then the argument t_2 should have type θ_i for every $i\in\{1,\ldots,n\}$. In the rule, n may be 0, in which case t_2 need not be typed. For example, $x:\top\to q_0\vdash_{\mathcal{B}} x\,(z\,z):q_0$ holds.

We now define typing for recursion schemes.

Rewriting rules of recursion scheme $\mathcal{G}_0$:

$$S \to F\ \mathtt{c} \qquad F\ x \to \mathtt{br}\ x\ (\mathtt{a}(F(\mathtt{b}(x))))$$

Transition rules of automaton $\mathcal{B}_0$:

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

Fig. 6. A running example: recursion scheme $\mathcal{G}_0$ (Example 2.1) and trivial automaton $\mathcal{B}_0$ (Example 2.3).

Definition 4.4 *(typing for higher-order recursion schemes).* Let $\mathcal{B} = (\Sigma, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_{\mathcal{B},0})$ be a trivial automaton and $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ a higher-order recursion scheme. We write $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ if (i) $\Gamma ::_{\mathcal{B}} \mathcal{N}$ and (ii) $\Gamma \vdash_{\mathcal{B}} \mathcal{R}(F) : \theta_i$ holds for every $F : \theta_i \in \Gamma$. $(\mathcal{G}, t)$ *has type* θ *under* Γ, written $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, t) : \theta$ if $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ and $\Gamma \vdash_{\mathcal{B}} t : \theta$. We write $\vdash_{\mathcal{B}} (\mathcal{G}, t) : \theta$ if $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, t) : \theta$ for some Γ. A recursion scheme $\mathcal{G}$ is *well-typed* if $\vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ holds.

The first condition $\Gamma ::_{\mathcal{B}} \mathcal{N}$ for $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ above says that the type environment Γ for non-terminals should be a refinement of sort environment $\mathcal{N}$. For a given sort environment $\mathcal{N}$, there are only finitely many Γ that satisfy $\Gamma ::_{\mathcal{B}} \mathcal{N}$. Thus, the condition ensures that, for type inference, it suffices to search a valid type environment from only a finite number of candidates.

Example 4.1. Recall the recursion scheme $\mathcal{G}_0$ in Example 2.1 and the automaton $\mathcal{B}_0$ in Example 2.3, which have the rewriting rules $\mathcal{R}$ and the transition function δ shown in Figure 6. We shall use the model checking problem of whether $[\![\mathcal{G}_0]\!]$ is accepted by ${\mathcal{B}_0}^{\perp}$ as a running example in the present and next sections. Let Γ be $\{S : q_0, F : q_0 \wedge q_1 \to q_0\}$. Then, we have

$$\Gamma \vdash_{\mathcal{B}_0} F\ \mathtt{c} : q_0 \qquad\qquad \Gamma \vdash_{\mathcal{B}_0} \lambda x.(\mathtt{br}\ x\ (\mathtt{a}(F(\mathtt{b}(x))))) : q_0 \wedge q_1 \to q_0.$$

The former is derived by:

$$\frac{\Gamma \vdash_{\mathcal{B}_0} F : q_0 \wedge q_1 \to q_0 \quad \Gamma \vdash_{\mathcal{B}_0} \mathtt{c} : q_0 \quad \Gamma \vdash_{\mathcal{B}_0} \mathtt{c} : q_1}{\Gamma \vdash_{\mathcal{B}_0} F\ \mathtt{c} : q_0}$$

The latter is derived by:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash_{\mathcal{B}_0} \mathtt{br} : q_0 \to q_0 \to q_0 \quad \Gamma_1 \vdash_{\mathcal{B}_0} x : q_0}{\Gamma_1 \vdash_{\mathcal{B}_0} \mathtt{br}\ x : q_0 \to q_0} \quad \dfrac{\ldots}{\Gamma_1 \vdash_{\mathcal{B}_0} \mathtt{a}(F(\mathtt{b}(x))) : q_0}}{\Gamma_1 \vdash_{\mathcal{B}_0} \mathtt{br}\ x\ (\mathtt{a}(F(\mathtt{b}(x)))) : q_0}}{\Gamma \vdash_{\mathcal{B}_0} \lambda x.(\mathtt{br}\ x\ (\mathtt{a}(F(\mathtt{b}(x))))) : q_0 \wedge q_1 \to q_0}$$

where $\Gamma_1 = \Gamma \cup \{x : q_0, x : q_1\}$. Thus, we have $\Gamma \vdash_{\mathcal{B}_0} (\mathcal{G}_0, S) : q_0$. □

Remark 4.1. The type system above can be easily extended to deal with *non-deterministic* trivial automata [Aehlig 2007] $(\Sigma_{\mathcal{B}}, Q_{\mathcal{B}}, \Delta_{\mathcal{B}}, q_{\mathcal{B},0})$ where $\Delta_{\mathcal{B}}$ is a map from $Q \times dom(\Sigma)$ to 2^{Q^*}. It suffices to replace rule T-Const with the following rule:

$$\frac{\Delta_{\mathcal{B}}(q, a) \ni q_1 \cdots q_n}{\Gamma \vdash_{\mathcal{B}} a : q_1 \to \cdots \to q_n \to q}$$

4.2 Soundness and Completeness

We use syntactic proof techniques to show that the type system above is sound and complete with respect to the model checking problem, i.e., that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^\perp$ if and only if $\mathcal{G}$ is well-typed in the type system. The meta-variables t and u range over applicative terms (that do not contain λ-abstractions) below.

We first prove the soundness.

THEOREM 4.1 (SOUNDNESS). *Let $\mathcal{G}$ be a recursion scheme and $\mathcal{B} = (\Sigma_\mathcal{B}, Q_\mathcal{B}, \delta_\mathcal{B}, q_{\mathcal{B},0})$ a trivial automaton. If $\vdash_\mathcal{B} (\mathcal{G}, S) : q_{\mathcal{B},0}$, then $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^\perp$.*

The proof of the theorem above is similar to standard syntactic proofs of type soundness [Wright and Felleisen 1994]. We first prove that typing is preserved by substitutions and reductions.

LEMMA 4.2 (WEAKENING). *If $\Gamma' \vdash_\mathcal{B} t : \theta$ and $\Gamma' \subseteq \Gamma$, then $\Gamma \vdash_\mathcal{B} t : \theta$.*

PROOF. This follows by straightforward induction on the derivation of $\Gamma' \vdash_\mathcal{B} t : \theta$. □

LEMMA 4.3 (SUBSTITUTION). *If $\Gamma \cup \{x : \theta_1, \ldots, x : \theta_m\} \vdash_\mathcal{B} t : \theta$ and $\Gamma \vdash_\mathcal{B} u : \theta_i$ for every $i \in \{1, \ldots, m\}$ with $x \notin dom(\Gamma)$, then $\Gamma \vdash_\mathcal{B} [u/x]t : \theta$.*

PROOF. This follows by straightforward induction on the derivation of $\Gamma \cup \{x : \theta_1, \ldots, x : \theta_m\} \vdash_\mathcal{B} t : \theta$. □

LEMMA 4.4 (TYPE PRESERVATION). *If $\vdash_\mathcal{B} (\mathcal{G}, t) : q$ and $t \longrightarrow_\mathcal{G} t'$, then $\vdash_\mathcal{B} (\mathcal{G}, t') : q$.*

PROOF. Suppose $\vdash_\mathcal{B} (\mathcal{G}, t) : q$, i.e., there exists Γ such that $\Gamma \vdash_\mathcal{B} (\mathcal{G}, t) : q$.

It suffices to show $\Gamma \vdash_\mathcal{B} t' : q$ by induction on the derivation of $t \longrightarrow_\mathcal{G} t'$. As the induction step is easy, we show only the base case, where $t = F\, u_1 \cdots u_k$ and $t' = [u_1/x_1, \ldots, u_k/x_k]s$ with $\mathcal{R}_\mathcal{G}(F) = \lambda x_1. \cdots \lambda x_k.s$. By the condition $\Gamma \vdash t : q$, we have:

$$(F : \textstyle\bigwedge_{j \in \{1,\ldots,n_1\}} \theta_{1,j} \to \cdots \to \bigwedge_{j \in \{1,\ldots,n_k\}} \theta_{k,j} \to q) \in \Gamma$$
$$\Gamma \vdash_\mathcal{B} u_i : \theta_{i,j} \text{ (for each } i \in \{1, \ldots, k\}, j \in \{1, \ldots, n_i\})$$

By the condition $\Gamma \vdash_\mathcal{B} (\mathcal{G}, t) : \theta$, it must be the case that:

$$\Gamma \vdash_\mathcal{B} \lambda x_1. \cdots \lambda x_k.s : \textstyle\bigwedge_{j \in \{1,\ldots,n_1\}} \theta_{1,j} \to \cdots \to \bigwedge_{j \in \{1,\ldots,n_k\}} \theta_{k,j} \to q,$$

which implies

$$\Gamma \cup \{x_i : \theta_{i,j} \mid i \in \{1, \ldots, k\}, j \in \{1, \ldots, n_i\}\} \vdash s : q.$$

By applying Lemma 4.3, we obtain $\Gamma \vdash [u_1/x_1, \ldots, u_k/x_k]s : q$ (i.e. $\Gamma \vdash t' : q$) as required. □

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 $t^\perp$ (recall Definition 2.5).

LEMMA 4.5. *If $\Gamma \vdash_\mathcal{B} t : q$, then $t^\perp$ is accepted by $\mathcal{B}^\perp$ 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 must be of the form $a\ t_1\ \cdots\ t_n$ with ${t_i}^\perp = t'_i$ for $i = 1, \ldots, n$. By $\Gamma \vdash_{\mathcal{B}} t : q$, we have $\Gamma \vdash_{\mathcal{B}} a : q_1 \to \cdots \to q_n \to q$ and $\Gamma \vdash_{\mathcal{B}} t_i : q_i$ $(i = 1, \ldots, n)$ for some $q_1, \ldots, q_n$. By the induction hypothesis, ${t_i}^\perp$ must be accepted by $\mathcal{B}^\perp$ from state q_i. $\Gamma \vdash_{\mathcal{B}} a : q_1 \to \cdots \to q_n \to q$ implies that $\delta_{\mathcal{B}}(q, a) = q_1 \cdots q_n$. Thus, $t^\perp$ must be accepted by $\mathcal{B}^\perp$ from state q. □

We are now ready to prove Theorem 4.1.

PROOF OF THEOREM 4.1. Suppose $\vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$. Since $[\![\mathcal{G}]\!] = \bigsqcup\{t^\perp \mid S \longrightarrow^*_{\mathcal{G}} t\}$, it suffices to show that $t^\perp$ is accepted by $\mathcal{B}^\perp$ for every t such that $S \longrightarrow^*_{\mathcal{G}} t$. (In fact, if R_t is a run-tree of $\mathcal{B}^\perp$ over $t^\perp$, then $\bigcup_{S \longrightarrow^*_{\mathcal{G}} t} R_t$ is a run-tree of $\mathcal{B}^\perp$ over $[\![\mathcal{G}]\!]$.) If $S \longrightarrow^*_{\mathcal{G}} t$, then by Lemma 4.4, we have $\vdash_{\mathcal{B}} (\mathcal{G}, t) : q_{\mathcal{B},0}$, which implies that $\Gamma \vdash_{\mathcal{B}} t : q_{\mathcal{B},0}$ holds for some Γ. By Lemma 4.5, $t^\perp$ is accepted by $\mathcal{B}^\perp$. □

Next, we prove the completeness of the type system.

THEOREM 4.6 (COMPLETENESS). *Let $\mathcal{G}$ be a recursion scheme and $\mathcal{B} = (\Sigma_{\mathcal{B}}, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_{\mathcal{B},0})$ be a trivial automaton. If $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^\perp$, then $\vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$.*

We prepare a few lemmas before proving the theorem. We first note the relation between sort assignment and type assignment (c.f. Definition 2.3 and Definition 4.3).

LEMMA 4.7. *Let $\mathcal{B} = (\Sigma_{\mathcal{B}}, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_{\mathcal{B},0})$ be a trivial automaton. If $\mathcal{K} \vdash_{\Sigma_{\mathcal{B}}} t : \kappa$ and $\Gamma \vdash_{\mathcal{B}} t : \theta$ with $\Gamma ::_{\mathcal{B}} \mathcal{K}$, then $\theta \in \mathbf{ATypes}_{\kappa,\mathcal{B}}$.*

PROOF. This follows by straightforward induction on the structure of t. □

Next, we show that typing is preserved by the *inverse* of substitutions and reductions (c.f. Lemmas 4.3 and 4.4).

LEMMA 4.8 (INVERSE SUBSTITUTION). *Suppose $\mathcal{K} \vdash_{\Sigma_{\mathcal{B}}} u : \kappa$ and $\Gamma ::_{\mathcal{B}} \mathcal{K}$. If $\Gamma \vdash_{\mathcal{B}} [u/x]t : \theta$ and $x \notin dom(\Gamma)$, then there exist $n\,(\geq 0)$ and $\theta_1, \ldots, \theta_n \in \mathbf{ATypes}_{\kappa,\mathcal{B}}$ such that $\Gamma \cup \{x : \theta_1, \ldots, x : \theta_n\} \vdash_{\mathcal{B}} t : \theta$ and $\Gamma \vdash_{\mathcal{B}} u : \theta_i$ for each $i \in \{1, \ldots, n\}$.*

PROOF. This follows by induction on the structure of t. If t is a terminal or a variable $y(\neq x)$, then $[u/x]t = t$. Thus, the required condition holds for $n = 0$. If $t = x$, then we have $\Gamma \vdash_{\mathcal{B}} u : \theta$. The required condition holds for $n = 1$ and $\theta_1 = \theta$. Note that we have $\theta \in \mathbf{ATypes}_{\kappa,\mathcal{B}}$ by Lemma 4.7. If $t = t_1 t_2$, then $[u/x]t = ([u/x]t_1)([u/x]t_2)$. By the condition $\Gamma \vdash_{\mathcal{B}} [u/x]t : \theta$, we have:

$$\Gamma \vdash_{\mathcal{B}} [u/x]t_1 : \textstyle\bigwedge_{i \in \{1,\ldots,m\}} \theta'_i \to \theta$$
$$\Gamma \vdash_{\mathcal{B}} [u/x]t_2 : \theta'_i \text{ (for each } i \in \{1, \ldots, m\})$$

where m may be 0. By the induction hypothesis, we have:

$$\Gamma \cup \{x : \theta_{0,1}, \ldots, x : \theta_{0,\ell_0}\} \vdash_{\mathcal{B}} t_1 : \textstyle\bigwedge_{i \in \{1,\ldots,m\}} \theta'_i \to \theta$$
$$\Gamma \cup \{x : \theta_{i,1}, \ldots, x : \theta_{i,\ell_i}\} \vdash_{\mathcal{B}} t_2 : \theta'_i \text{ for each } i \in \{1, \ldots, m\}$$
$$\Gamma \vdash_{\mathcal{B}} u : \theta_{i,j} \text{ and } \theta_{i,j} \in \mathbf{ATypes}_{\kappa,\mathcal{B}} \text{ for each } i \in \{0, \ldots, m\}, j \in \{1, \ldots, \ell_i\}$$

Let $\{\theta_1, \ldots, \theta_n\}$ be the set $\{\theta_{i,j} \mid i \in \{0, \ldots, m\}, j \in \{1, \ldots, \ell_i\}\}$ and n be the size of the set. We can use Lemma 4.2 to derive $\Gamma \cup \{x : \theta_1, \ldots, x : \theta_n\} \vdash_{\mathcal{B}} t_1 : \bigwedge_{i \in \{1,\ldots,m\}} \theta'_i \to \theta$ and $\Gamma \cup \{x : \theta_1, \ldots, x : \theta_n\} \vdash_{\mathcal{B}} t_2 : \theta'_i$ for each $i \in \{1, \ldots, m\}$, from which $\Gamma \cup \{x : \theta_1, \ldots, x : \theta_n\} \vdash_{\mathcal{B}} t_1 t_2 : \theta$ follows. Thus we have the required conditions. □

Lemma 4.9 (type preservation by inverse reduction). *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ be a recursion scheme and $\mathcal{B}$ be a trivial automaton. If $\vdash_{\mathcal{B}} (\mathcal{G}, t') : q$ and $t \longrightarrow_{\mathcal{G}} t'$ with $\mathcal{N} \vdash_{\Sigma} t : o$, then $\vdash_{\mathcal{B}} (\mathcal{G}, t) : q$.*

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 $t = F\, u_1 \cdots u_k$ and $t' = [u_1/x_1, \ldots, u_k/x_k]s$, with $\mathcal{R}_{\mathcal{G}}(F) = \lambda x_1. \cdots \lambda x_k.s$. We have:

$$\begin{array}{l} \mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_k \to o \\ \mathcal{N} \vdash_{\Sigma_{\mathcal{B}}} u_i : \kappa_i \text{ for each } i \in \{1, \ldots, k\} \end{array}$$

Suppose $\vdash_{\mathcal{B}} (\mathcal{G}, t') : q$, i.e., $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, t') : q$ for some Γ. By $\Gamma \vdash_{\mathcal{B}} t' : q$ and Lemma 4.8, we have:

$$\begin{array}{l} \Gamma \cup \{x_i : \theta_{i,j} \mid i \in \{1, \ldots, k\}, j \in \{1, \ldots, n_i\}\} \vdash_{\mathcal{B}} s : q \\ \Gamma \vdash_{\mathcal{B}} u_i : \theta_{i,j} \text{ for each } i \in \{1, \ldots, k\}, j \in \{1, \ldots, n_i\}. \\ \theta_{i,j} \in \mathbf{ATypes}_{\kappa_i, \mathcal{B}} \end{array}$$

Let θ' be $\bigwedge_{j \in \{1,\ldots,n_1\}} \theta_{1,j} \to \cdots \bigwedge_{j \in \{1,\ldots,n_k\}} \theta_{k,j} \to q$ and Γ' be $\Gamma \cup \{F : \theta'\}$. Then we have $\theta' \in \mathbf{ATypes}_{\mathcal{N}(F), \mathcal{B}}$ and $\Gamma' ::_{\mathcal{B}} \mathcal{N}$, with $\Gamma' \vdash_{\mathcal{B}} t : q$ and $\Gamma' \vdash_{\mathcal{B}} \lambda \widetilde{x}.s : \theta'$. Thus, we have $\Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, t) : q$ as required. □

Following is a converse property of Lemma 4.5.

Lemma 4.10. *If $t^{\perp}$ is accepted by $\mathcal{B}^{\perp}$ from state q, then $\emptyset \vdash_{\mathcal{B}^{\perp}} t^{\perp} : q$.*

Proof. The proof proceeds by induction on the structure of $t^{\perp}$. If $t^{\perp} = \perp$, the result follows immediately: Note that, by the definition of $\mathcal{B}^{\perp}$ (Definition 2.6) and rule T-Const, $\emptyset \vdash_{\mathcal{B}^{\perp}} \perp : q$ holds for every q. 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 $\mathcal{B}^{\perp}$ from state q, there must be states $q_1, \ldots, q_n$ such that (i) $\delta_{\mathcal{B}}(q, a) = q_1 \cdots q_n$, and (ii) $t'_1, \ldots, t'_n$ are accepted by $\mathcal{B}^{\perp}$ from states $q_1, \ldots, q_n$ respectively. By the induction hypothesis and condition (ii), we have $\emptyset \vdash_{\mathcal{B}^{\perp}} t'_i : q_i$ for each i. The condition (i) implies $\emptyset \vdash_{\mathcal{B}} a : q_1 \to \cdots \to q_n \to q$. Thus, we have $\emptyset \vdash_{\mathcal{B}^{\perp}} t^{\perp} : q$ as required. □

Next, we give a fixed-point characterization of the typability of a recursion scheme.

Definition 4.5. Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ be a recursion scheme and $\mathcal{B} = (\Sigma, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_{\mathcal{B},0})$ be a trivial automaton. We define the set $\mathcal{TE}_{\mathcal{G},\mathcal{B}}$ of type environments by:

$$\mathcal{TE}_{\mathcal{G},\mathcal{B}} = \{\Gamma \mid \Gamma ::_{\mathcal{B}} \mathcal{N}\}.$$

We write $\Gamma_{\mathbf{max}}$ for the greatest element of $\mathcal{TE}_{\mathcal{G},\mathcal{B}}$, i.e., $\{F : \theta \mid F \in dom(\mathcal{N}), \theta \in \mathbf{ATypes}_{\mathcal{N}(F),\mathcal{B}}\}$.

We define a map $\mathcal{F}_{\mathcal{G},\mathcal{B}}$ from $\mathcal{TE}_{\mathcal{G},\mathcal{B}}$ to $\mathcal{TE}_{\mathcal{G},\mathcal{B}}$ by:

$$\mathcal{F}_{\mathcal{G},\mathcal{B}}(\Gamma) = \{F : \theta \in \Gamma \mid \Gamma \vdash_{\mathcal{B}} \mathcal{R}(F) : \theta\}.$$

Note that, for given $\mathcal{G}$ and $\mathcal{B}$, $\mathcal{TE}_{\mathcal{G},\mathcal{B}}$ and $\Gamma_{\mathbf{max}}$ defined above are finite sets. The function $\mathcal{F}_{\mathcal{G},\mathcal{B}}$ checks, for each type assumption $F : \theta \in \Gamma$, whether the body of F indeed has type θ, and removes $F{:}\theta$ from Γ if that is not the case. $\mathcal{F}_{\mathcal{G},\mathcal{B}}$ is monotonic (i.e., $\Gamma_1 \subseteq \Gamma_2$ implies $\mathcal{F}_{\mathcal{G},\mathcal{B}}(\Gamma_1) \subseteq \mathcal{F}_{\mathcal{G},\mathcal{B}}(\Gamma_2)$) and contractive (i.e., $\Gamma \supseteq \mathcal{F}_{\mathcal{G},\mathcal{B}}(\Gamma)$ for every Γ).

The following lemma gives a characterization of the typability in terms of a fixed-point of $\mathcal{F}_{\mathcal{G},\mathcal{B}}$.

LEMMA 4.11. *Let Γ be an element of $\mathcal{TE}_{\mathcal{G},\mathcal{B}}$. Then, there exists $m(\leq |\Gamma|)$ such that $\mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma)$ is a fixed-point of $\mathcal{F}_{\mathcal{G},\mathcal{B}}$, i.e., $\mathcal{F}_{\mathcal{G},\mathcal{B}}(\mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma)) = \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma)$. Furthermore, $\mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma)$ is the largest Γ' such that $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma'$ and $\Gamma' \subseteq \Gamma$. In particular, if $\Gamma = \Gamma_{\mathbf{max}}$, then $\vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ if and only if $S : q_{\mathcal{B},0} \in \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma)$.*

PROOF. The first part is a special case of Knaster's fixed-point theorem [Knaster 1927], restricted to functions on finite sets: Since $\mathcal{F}_{\mathcal{G},\mathcal{B}}$ is monotonic and contractive, $\mathcal{F}^0_{\mathcal{G},\mathcal{B}}(\Gamma), \mathcal{F}^1_{\mathcal{G},\mathcal{B}}(\Gamma), \mathcal{F}^2_{\mathcal{G},\mathcal{B}}(\Gamma), \ldots$ is a decreasing sequence consisting of subsets of Γ, which must converge in at most $|\Gamma|$ steps.

For the second part, we first note that $\vdash_{\mathcal{B}} \mathcal{G} : \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma)$ and $\mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma) \subseteq \Gamma$ follow immediately from $\mathcal{F}_{\mathcal{G},\mathcal{B}}(\mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma)) = \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma)$ and the fact that $\mathcal{F}_{\mathcal{G},\mathcal{B}}$ is contractive. Suppose $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma''$ and $\Gamma'' \subseteq \Gamma$. From the first condition, we have $\Gamma'' \subseteq \mathcal{F}_{\mathcal{G},\mathcal{B}}(\Gamma'')$. Thus, by the monotonicity of $\mathcal{F}_{\mathcal{G},\mathcal{B}}$, we have:

$$\Gamma'' \subseteq \mathcal{F}_{\mathcal{G},\mathcal{B}}(\Gamma'') \subseteq \cdots \subseteq \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma'') \subseteq \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma).$$

If $\Gamma = \Gamma_{\mathbf{max}}$ and $\vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$, then there exists Γ_1 such that $S : q_{\mathcal{B},0} \in \Gamma_1$ and $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma_1$, which implies $S : q_{\mathcal{B},0} \in \Gamma_1 \subseteq \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}})$. The converse is trivial. □

We are now ready to prove Theorem 4.6.

PROOF OF THEOREM 4.6. Suppose that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^\perp$. By Lemma 4.11, there exists m such that $\mathcal{F}^{m+1}_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}}) = \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}})$.

From $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$, we define a higher-order recursion scheme $\mathcal{G}^{(m)}$ without recursion by:

$$\begin{array}{rcl}
\mathcal{G}^{(m)} & = & (\Sigma^\perp, \mathcal{N}^{(m)}, \mathcal{R}^{(m)}, S^{(m)}) \\
\mathcal{N}^{(m)} & = & \{F^{(j)} : \kappa \mid F : \kappa \in \mathcal{N}, j \in \{0, \ldots, m\}\} \\
\mathcal{R}^{(m)} & = & \{F^{(j)}\, \widetilde{x} \to t^{(j-1)} \mid F\, \widetilde{x} \to t \in \mathcal{R}, j \in \{1, \ldots, m\}\} \\
 & & \cup \{F^{(0)}\, \widetilde{x} \to \perp \mid F\, \widetilde{x} \to t \in \mathcal{R}\}
\end{array}$$

Here, $t^{(j)}$ is the term obtained from t by replacing each non-terminal F' with $F'^{(j)}$. By the definition, it is easy to show by induction on m that $[\![\mathcal{G}^{(m)}]\!]$ is an approximation of $[\![\mathcal{G}]\!]$, i.e. $[\![\mathcal{G}^{(m)}]\!] \sqsubseteq [\![\mathcal{G}]\!]$ (recall Definition 2.1). So, $[\![\mathcal{G}^{(m)}]\!]$ is accepted by $\mathcal{B}^\perp$. Because $\mathcal{G}^{(m)}$ does not contain recursion, by the strong normalization of the simply-typed λ-calculus, $[\![\mathcal{G}^{(m)}]\!]$ must be a finite tree and $S \longrightarrow^*_{\mathcal{G}^{(m)}} [\![\mathcal{G}^{(m)}]\!]$. Thus, by Lemma 4.10, we have $\vdash_{\mathcal{B}^\perp} [\![\mathcal{G}^{(m)}]\!] : q_{\mathcal{B},0}$. By Lemma 4.9, we have $\Gamma \vdash_{\mathcal{B}^\perp} (\mathcal{G}^{(m)}, S^{(m)}) : q_{\mathcal{B},0}$ for some Γ.

Let us define $\Gamma_0, \ldots, \Gamma_m$ by:

$$\Gamma_k = \{F : \theta \mid F^{(j)} : \theta \in \Gamma, j \geq k\}.$$

We first show that $\Gamma_{k+1} \subseteq \mathcal{F}_{\mathcal{G},\mathcal{B}^\perp}(\Gamma_k)$ holds for each $k \in \{0, \ldots, m-1\}$. Suppose $F : \theta \in \Gamma_{k+1}$. Then, we have $F^{(j)} : \theta \in \Gamma$ and $\Gamma \vdash_{\mathcal{B}^\perp} \mathcal{R}^{(m)}(F^{(j)}) : \theta$ for some $j \geq k+1$. Since $\mathcal{R}^{(m)}(F^{(j)})$ contains only non-terminals of the form $F'^{(j-1)}$, we have $\Gamma'_k \vdash_{\mathcal{B}^\perp} \mathcal{R}^{(m)}(F^{(j)}) : \theta$ for $\Gamma'_k = \{F'^{(\ell)} : \theta' \in \Gamma \mid \ell \geq k\}$. By renaming each $F'^{(\ell)}$ to F' in the derivation of $\Gamma'_k \vdash_{\mathcal{B}^\perp} \mathcal{R}^{(m)}(F^{(j)}) : \theta$, we obtain $\Gamma_k \vdash_{\mathcal{B}^\perp} \mathcal{R}(F) : \theta$. Thus, we have $F : \theta \in \mathcal{F}_{\mathcal{G},\mathcal{B}^\perp}(\Gamma_k)$.

By the above property and the monotonicity of $\mathcal{F}_{\mathcal{G},\mathcal{B}^\perp}$, we have:

$$\Gamma_m \subseteq \mathcal{F}_{\mathcal{G},\mathcal{B}^\perp}(\Gamma_{m-1}) \subseteq \cdots \subseteq \mathcal{F}^m_{\mathcal{G},\mathcal{B}^\perp}(\Gamma_0) \subseteq \mathcal{F}^m_{\mathcal{G},\mathcal{B}^\perp}(\Gamma_{\mathbf{max}}) = \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}}).$$

(The rightmost equality uses the fact that $\perp$ is not contained in $\mathcal{G}$.) Thus, $S : q_{\mathcal{B},0} \in \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}})$. By Lemma 4.11, we have $\vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$ as required. □

4.3 A Naive Type Checking Algorithm and Complexity Results

Lemma 4.11 gives the following type checking algorithm:

(1) Compute $\mathcal{F}^1_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}}), \mathcal{F}^2_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}}), \ldots$, and find m such that either $\mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}}) = \mathcal{F}^{m+1}_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}})$ or $S : q_{\mathcal{B},0} \notin \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}})$.
(2) Answer whether $S : q_{\mathcal{B},0} \in \mathcal{F}^m_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}})$.

The first step must terminate by Lemma 4.11, and m is bounded by $|\Gamma_{\mathbf{max}}| = \sum_{F \in dom(\mathcal{N})} |\mathbf{ATypes}_{\mathcal{N}(F),\mathcal{B}}|$.

Example 4.2. Consider the running example in Figure 6.

$$\begin{aligned}\Gamma_{\mathbf{max}} = \{&S : q_0, S : q_1, F : \top \to q_0, F : \top \to q_1, F : q_0 \to q_0, F : q_0 \to q_1,\\ &F : q_1 \to q_0, F : q_1 \to q_1, F : q_0 \wedge q_1 \to q_0, F : q_0 \wedge q_1 \to q_1\}.\end{aligned}$$

By repeated applications of $\mathcal{F}_{\mathcal{G}_0,\mathcal{B}_0}$, we obtain:

$$\begin{aligned}\mathcal{F}_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_{\mathbf{max}}) &= \{S : q_0, S : q_1, F : q_0 \to q_0, F : q_0 \wedge q_1 \to q_0\}\\ \mathcal{F}^2_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_{\mathbf{max}}) &= \{S : q_0, F : q_0 \wedge q_1 \to q_0\}\\ \mathcal{F}^3_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_{\mathbf{max}}) &= \{S : q_0, F : q_0 \wedge q_1 \to q_0\} = \mathcal{F}^2_{\mathcal{G}_0,\mathcal{B}_0}(\Gamma_{\mathbf{max}})\end{aligned}$$

Thus, $\{S : q_0, F : q_0 \wedge q_1 \to q_0\}$ is the greatest fixed-point. Since it contains $S : q_0$, we know $\vdash_{\mathcal{B}_0} (\mathcal{G}_0, S) : q_0$, i.e. $[\![\mathcal{G}_0]\!]$ is accepted by ${\mathcal{B}_0}^\perp$. □

We now discuss the time complexity of the above algorithm. Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ be an order-k higher-order recursion scheme, and $\mathcal{B}$ be a deterministic trivial automaton $\mathcal{B} = (\Sigma, Q, \delta, q_0)$. Let $|\mathcal{G}|$ be the size of $\mathcal{G}$, i.e., $\sum_{F \in dom(\mathcal{N})} \mathit{size}(\mathcal{R}(F))$ (where the size $\mathit{size}(t)$ of a term t is defined as usual). Let A be the largest arity of terminals, non-terminals, and variables in $\mathcal{G}$. We shall show below that the naive algorithm given above runs in time $O(|\mathcal{G}|^2 \mathbf{exp}_k((A \times |Q|)^{1+\epsilon}))$ for any $\epsilon > 0$ if $k\,(\geq 1)$ is fixed.

We first restrict the shape of the recursion scheme $\mathcal{G}$. A recursion scheme is in *normal form* if each rule of the recursion scheme is of the form $F\,\widetilde{x} \to f(f_1\widetilde{x}_1)\cdots(f_\ell\widetilde{x}_\ell)$, where $f, f_1, \ldots, f_\ell$ are non-terminals, terminals, or variables. (Note that ℓ can be 0.) We assume below that the recursion scheme $\mathcal{G}$ is in normal form. Otherwise, $\mathcal{G}$

can be transformed to a normal form as follows. For a rule $F\,\widetilde{x} \to f\,t_1\,\cdots\,t_\ell$ where t_i is not of the form $f_i\,\widetilde{x}_i$, replace the body of F with $f\,t_1\,\cdots\,t_{i-1}\,(G\,\widetilde{x}_i)\,t_{i+1}\,\cdots\,t_\ell$ and add the rule $G\ \widetilde{x}_i\ \widetilde{y} \to t_i\,\widetilde{y}$, where G is a fresh non-terminal symbol, and $\{\widetilde{x}_i\}(\subseteq\{\widetilde{x}\})$ is the set of variables in t_i. Repeated application of this transformation yields a recursion scheme in normal form. The order, size, and largest arity of the resulting recursion scheme (in normal form) $\mathcal{G}'$ can be estimated as follows.

(1) The order of $\mathcal{G}'$ is the same as that of $\mathcal{G}$.
(2) The size of $\mathcal{G}'$ is $O((1+A)\times|\mathcal{G}|)$. (Note that each transformation step increases the size of a recursion scheme only by $O(1+A)$.)
(3) The largest arity of $\mathcal{G}'$ is at most $2\times A$. (Note that the arity of the fresh non-terminal symbol G introduced above is bounded by $|\widetilde{x}_i|+|\widetilde{y}|$, where $|\widetilde{x}_i|$ and $|\widetilde{y}|$ are both bounded by A.)

Thus, the transformation does not affect the time complexity $O(|\mathcal{G}|^2\mathbf{exp}_k((A\times|Q|)^{1+\epsilon}))$.

Next, we estimate the size of the set $\mathbf{ATypes}_{\kappa,\mathcal{B}}$. If κ has order k, $|\mathbf{ATypes}_{\kappa,\mathcal{B}}|$ is bounded by a_k, given by:

$$a_0 = |Q| \qquad a_{k+1} = \underbrace{2^{a_k}\times\cdots\times 2^{a_k}}_{A}\times a_k = 2^{A\times a_k\log a_k}$$

Thus, $a_k = O(\mathbf{exp}_k((A\times|Q|)^{1+\epsilon}))$ for any $\epsilon>0$.

We now estimate the cost for computing $\mathcal{F}_{\mathcal{G},\mathcal{B}}(\Gamma)$. For each $F:\theta\in\Gamma$, whether $\Gamma\vdash_{\mathcal{B}}\mathcal{R}(F):\theta$ holds can be checked as follows. By the assumption that the recursion scheme is in normal form, $\mathcal{R}(F)$ must be of the form $\lambda\widetilde{x}.f(f_1\widetilde{x}_1)\cdots(f_\ell\widetilde{x}_\ell)$. Thus, θ is of the form $\bigwedge_{j\in J_1}\theta_{1,j}\to\cdots\to\bigwedge_{j\in J_\ell}\theta_{\ell,j}\to q$. We just need to enumerate the set S of all the possible types of $f(f_1\widetilde{x}_1)\cdots(f_\ell\widetilde{x}_\ell)$ under $\Gamma\cup\{x_i{:}\theta_{i,j}\mid i\in\{1,\ldots,\ell\},j\in J_i\}$, and check whether $q\in S$. Given the set of types of t_1 (whose size is at most a_k) and the set of types of t_2 (whose size is at most a_{k-1}), the set of types of t_1t_2 can be computed in time $O(a_k\times a_{k-1})$: For each type $\bigwedge_{j\in J}\theta_j\to\theta$ of t_1, it suffices to check whether the set of types of t_2 contains every θ_j.[6] As $f(f_1\widetilde{x}_1)\cdots(f_\ell\widetilde{x}_\ell)$ contains at most $2A$ applications, its types can be enumerated in time $2A\times O(a_k\times a_{k-1})=O(A\times a_k^2)$. Thus, $\Gamma\vdash_{\mathcal{B}}\mathcal{R}(F):\theta$ can be checked in time $O((1+A)\times a_k^2)$. As the size of Γ is at most $|\mathcal{G}|\times a_k$, the time cost for computing $\mathcal{F}_{\mathcal{G},\mathcal{B}}(\Gamma)$ is $O(|\mathcal{G}|\times(1+A)\times a_k^3)=O(|\mathcal{G}|\times\mathbf{exp}_k((A\times|Q|)^{1+\epsilon}))$.

Finally, the number m of iterations is bounded by $|\Gamma_{\mathbf{max}}|=|\mathbf{ATypes}_{\mathcal{N}(F_1),\mathcal{B}}|+\cdots+|\mathbf{ATypes}_{\mathcal{N}(F_n),\mathcal{B}}|=O(|\mathcal{G}|\times a_k)$. Thus, the algorithm runs in time $O(|\mathcal{G}|^2\times\mathbf{exp}_k((A\times|Q|)^{1+\epsilon}))$ for a fixed $k\,(\geq 1)$.

The algorithm above can actually be optimized to yield an $O(|\mathcal{G}|\times\mathbf{exp}_k((A\times|Q|)^{1+\epsilon}))$ algorithm (for $k\geq 1$), by using the standard method for solving constraints over finite semi-lattices [Rehof and Mogensen 1999]. (The idea is, at each iteration for computing $\mathcal{F}^j_{\mathcal{G},\mathcal{B}}(\Gamma_{\mathbf{max}})$, instead of recomputing $\Gamma_F=\{F:\theta\in\Gamma\mid\Gamma\vdash_{\mathcal{B}}\mathcal{R}(F):\theta\}$ for every F, to compute Γ_F only for F such that the types of the

[6]Here, we have assumed that a set is implemented as an array, and the membership is computed in time $O(1)$. For the overall complexity result, it suffices to just assume that set operations can be performed in time polynomial in the size of sets.

non-terminals in $\mathcal{R}(F)$ have changed in the previous step.) Thus, the algorithm runs in time linear in the size of the recursion scheme, provided that k, A and $|Q|$ are fixed. On the parameters A and $|Q|$, Kobayashi and Ong [2011] give tighter complexity bounds: recursion scheme model checking for the class of deterministic trivial automata is $(k-1)$-EXPTIME complete (recall Remark 2.1).

5. A HYBRID TYPE CHECKING ALGORITHM

The naive type checking algorithm described in Section 4.3 is impractical. Although it runs in time linear in the size of recursion schemes, the constant factor is $\mathbf{exp}_k((A \times |Q|)^{1+\epsilon})$, which is prohibitively large. This section therefore describes a more realistic type checking (i.e. model checking) algorithm for recursion schemes. The worst-case complexity of the new algorithm is actually even worse, but it is faster for typical inputs than the naive algorithm, as confirmed by the experiments described in Section 6.

For understanding how and why the new algorithm works better, it would be helpful to consider why the naive algorithm is so slow. As is clear from the discussion in Section 4.3, the huge constant factor of the naive algorithm comes from the size of the set $\mathbf{ATypes}_{\kappa,\mathcal{B}}$ of types. The following table shows the size of $\mathbf{ATypes}_{\kappa,\mathcal{B}}$ for various sorts.

sort κ	$\lvert\mathbf{ATypes}_{\kappa,\mathcal{B}}\rvert$ ($\lvert Q\rvert = 2$)	$\lvert\mathbf{ATypes}_{\kappa,\mathcal{B}}\rvert$ ($\lvert Q\rvert = 4$)
$\mathsf{o} \to \mathsf{o}$	$2^2 \times 2 = 8$	$2^4 \times 4 = 64$
$(\mathsf{o} \to \mathsf{o}) \to \mathsf{o}$	$2^8 \times 2 = 512$	$2^{64} \times 4 \approx 6.4 \times 10^{19}$
$((\mathsf{o} \to \mathsf{o}) \to \mathsf{o}) \to \mathsf{o}$	$2^{512} \times 2 = 2^{513} \approx 10^{154}$	$2^{2^{66}} \times 4 > 10^{10000000000000000000}$

Here, the second and third columns respectively show the cases for $|Q| = 2$ and $|Q| = 4$. The naive algorithm always starts the fixed-point computation $\Gamma, \mathcal{F}^1_{\mathcal{G},\mathcal{B}}(\Gamma), \mathcal{F}^2_{\mathcal{G},\mathcal{B}}(\Gamma), \ldots$ from $\Gamma = \Gamma_{\mathbf{max}}$ that contains all the possible types, and checks whether each type is valid at each iteration of the fixed-point computation. Thus, the naive algorithm cannot be run at least for recursion schemes of order 3 or higher.

The analysis above suggests that if the fixed-point computation can be started from a type environment Γ that is much smaller than $\Gamma_{\mathbf{max}}$, then the fixed-point may be efficiently computed. To keep the completeness, Γ should contain type bindings required for typing the given recursion scheme. Thus, for example, $\Gamma = \emptyset$ is inappropriate. The type environment Γ, however, need not necessarily contain all the possibly valid typings. For the running example in Figure 6, for example, we can exclude out $S : q_1$ from Γ beforehand, as we are only interested in whether S has type q_0. How can we find such Γ in general?

The key observation to address the question above is that types of a function describe how that function is used in the recursion scheme. Therefore, information about the types of a function can be obtained by actually reducing the recursion scheme, and observing how the function is used. Actually, the completeness proof of Kobayashi and Ong's type system for the modal μ-calculus model checking of recursion schemes [Kobayashi and Ong 2009] (which is an extension of the type system in Section 4) gives a "procedure" for extracting the types of each function from an *infinite* reduction sequence of a recursion scheme. By modifying the "procedure", we can obtain an algorithm for extracting *partial* information about the

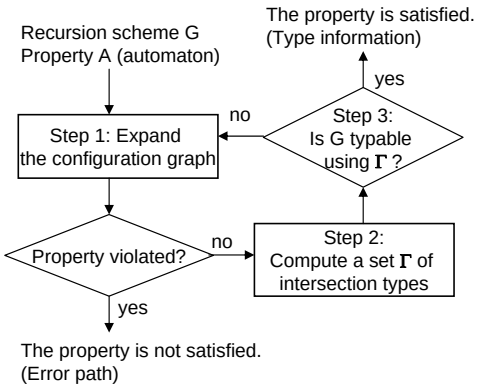


Fig. 7. The overview of the hybrid algorithm

types of functions from a *finite* reduction sequence of a recursion scheme. From the partial type information, we can construct a type environment Γ that is often much smaller than $\Gamma_{\mathbf{max}}$, and run the fixed-point computation from it.

The overall structure of the resulting algorithm (which we call a *hybrid* algorithm) is shown in Figure 7. As in the previous section, we fix a trivial automaton $\mathcal{B} = (\Sigma_{\mathcal{B}}, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_{\mathcal{B},0})$, and omit the subscript $\mathcal{B}$ when it is clear from the context. We also fix a recursion scheme $\mathcal{G} = (\Sigma_{\mathcal{B}}, \mathcal{N}, \mathcal{R}, S)$ below. First, we reduce (the initial symbol of) the recursion scheme $\mathcal{G}$ a finite number of steps and run the automaton $\mathcal{B}$ for the partially generated tree. A *configuration graph* is constructed (Step 1 in Figure 7), which represents how $\mathcal{G}$ has been reduced, and how the partially generated tree has been recognized by $\mathcal{B}$. If a property violation is found (i.e., the run of $\mathcal{B}$ over the partially generated tree gets stuck) during the reduction, then an error path is reported. Otherwise, a set Γ of type bindings is computed from the configuration graph (Step 2). In Step 3, it is checked whether the recursion scheme is typed by using only the types in Γ. If the recursion scheme is well-typed, then the property is satisfied (by the soundness of the type system, Theorem 4.1). Otherwise, go back to Step 1 and expand the configuration graph to get a larger set of 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.

Pseudo code for the overall algorithm is shown in Figure 8. We explain each step in more detail below.

5.1 Initialization and Step 1

In Step 1, we reduce the given recursion scheme a finite number of steps and construct a configuration graph.

Definition 5.1 *(configuration graphs).* A *configuration graph* is a labeled directed graph, where the label of each edge is a non-negative integer, and the label of each node is of the form $\langle t, q, f\rangle$. In $\langle t, q, f\rangle$, t is an applicative term of sort o, q is a

```
Init: (* Section 5.1 *)
  C := the initial configuration graph;
  Γ := ∅
  goto Step 1;

Step 1: (* Section 5.1 *)
  count := 0;
  while(count<MAX and an open node exists) do
  { N := an open node;
    if C can be expanded wrt N then {
      C := expand(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: (* Section 5.2 *)
  Γ := Γ ∪ ElimTE(Γ_C);
  goto Step 3;

Step 3: (* Section 5.3 *)
  while(Γ ≠ F_{G,B}(Γ)) do Γ := F_{G,B}(Γ); (* F_{G,B} is given in Definition 4.5 *)
  if S : q_{B,0} ∈ Γ then
    raise PROPERTY_SATISFIED(Γ)
  else
    goto Step 1;
```

Fig. 8. A hybrid model checking algorithm to check whether $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$

state of the automaton $\mathcal{B}$, and f is either `open` or `closed`. The *initial configuration graph* is a graph consisting of just a single node (called the *root node*), labeled by $\langle S, q_{\mathcal{B},0}, \texttt{open}\rangle$.

Let $\mathcal{C}$ be a configuration graph, and N a node of $\mathcal{C}$, labeled by $\langle t, q, \texttt{open}\rangle$. Then, an *expansion of* $\mathcal{C}$ *with respect to* N is the graph $\mathcal{C}'$ obtained from $\mathcal{C}$ by replacing the flag of N with `closed`, and adding nodes and (directed) edges as follows.

(1) Case $t = a\, t_1\, \cdots\, t_m$ and $\delta(q, a) = q_1 \cdots q_m$: For each $i \in \{1, \ldots, m\}$, (i) if a node labeled with $\langle t_i, q_i, f_i\rangle$ (for some f_i) does not exist, add a new node $\langle t_i, q_i, \texttt{open}\rangle$; and (ii) add a directed edge from N to the node labeled by $\langle t_i, q_i, f_i\rangle$ and label the edge with i.

(2) Case $t = F\, t_1\, \cdots\, t_m$ and $\mathcal{R}(F) = \lambda x_1. \cdots \lambda x_\ell.t$: Since t has sort `o`, we have $m = \ell$. 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 $\langle [t_1/x_1, \ldots, t_m/x_m]t, q, \texttt{open}\rangle$. 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 (a possibly 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 $\langle t, q, \texttt{open}\rangle$).

Recursion scheme $\mathcal{G}_0$:

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

Transition rules of automaton $\mathcal{B}_0$:

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

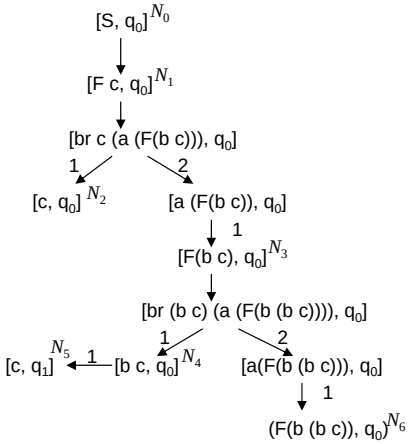


Fig. 9. A configuration graph for the running example (Figure 6). $N_0, N_1, \ldots$ are node names.

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 8 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, and 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 5.1. Figure 9 shows a configuration graph for the running example in Figure 6. The only open node is $(F(\mathtt{b}(\mathtt{b}\,\mathtt{c})), q_0)$. □

Example 5.2. Consider the recursion scheme $\mathcal{G}_2$ given by the following rewriting rules:

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

and the automaton $(\{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0\}, \{q_0, q_1\}, \delta, q_0)$ where

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

Figure 10 shows a configuration graph obtained by several expansions. 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`. □

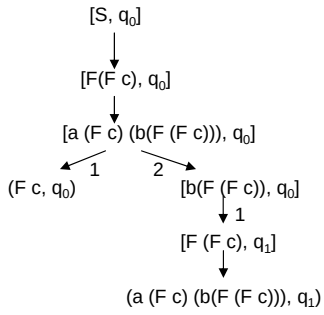


Fig. 10. A Configuration Graph Containing an Error Node

5.2 Step 2

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

As a configuration graph carries only partial type information, we extend the syntax of arrow types with type variables, which express unknown (sets of) types.

$$\tau \text{ (extended types)} ::= q \mid \bigwedge\{\tau_1,\ldots,\tau_m\} \to \tau \mid \bigwedge(\{\tau_1,\ldots,\tau_m\} \cup \alpha) \to \tau$$

By abuse of notation, we often write $\tau_1 \wedge \cdots \wedge \tau_m \to \tau$ and $\tau_1 \wedge \cdots \wedge \tau_m \wedge \alpha \to \tau$ for $\bigwedge\{\tau_1,\ldots,\tau_m\} \to \tau$ and $\bigwedge(\{\tau_1,\ldots,\tau_m\} \cup \alpha) \to \tau$ respectively.

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 [Kobayashi and Ong 2009]. 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. As defined below, for each node N labeled with $\langle t, q, f\rangle$, we assign a type $\tau_{t',N}$ to each prefix t' of t. We then define $\Gamma_{\mathcal{C}}$ as a collection of bindings $F : \tau_{F,N}$.

Definition 5.2 *(type environment $\Gamma_{\mathcal{C}}$).* Let $\mathcal{C}$ be a configuration graph, and let N be a node labeled with $\langle t, q, f\rangle$, and t' a prefix of t. The type $\tau_{t',N}$ is defined by induction on the sort of t':

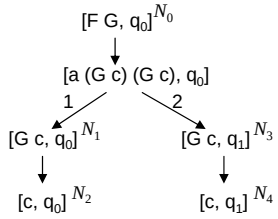
(1) Case t' has sort o: N must have a label of the form $\langle t', q, f\rangle$. Define $\tau_{t',N}$ as q.

(2) Case t' has sort $\kappa_1 \to \kappa_2$: 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 $\{N_1,\ldots,N_\ell\}$ 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 the occurrence of s_0 in 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 be $\{\tau_{s_0,N_1},\ldots,\tau_{s_0,N_\ell}\} \cup \alpha_{s_0,N}$ where $\alpha_{s_0,N}$ is a fresh type variable (which indicates that the type can be further refined by further expansion of the configuration graph). Otherwise, let S be $\{\tau_{s_0,N_1},\ldots,\tau_{s_0,N_\ell}\}$. Finally, define $\tau_{t',N}$ as $(\bigwedge S) \to \tau_{t'\,s_0,N}$. (Note that since the sorts of $t's_0$ and s_0 are κ_2 and κ_1 respectively, $\tau_{t'\,s_0,N}$ and τ_{s_0,N_i} are determined by the induction.)

Given a configuration graph $\mathcal{C}$, the type environment $\Gamma_{\mathcal{C}}$ is defined 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 5.3. Consider the following fragment of a configuration graph:



Here, superscripts $N_0, N_1, \ldots$ show the names of nodes. τ_{F,N_0} is computed as follows.

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

□

The type environment $\Gamma_{\mathcal{C}}$ above may contain type variables, which are eliminated by the procedure *ElimTE* below. As we will see later (in Section 5.4), if $\mathcal{G}$ is typable and a finite configuration graph $\mathcal{C}$ is sufficiently large, then $\mathit{ElimTE}(\Gamma_{\mathcal{C}})$ contains all the type bindings required to type $\mathcal{G}$.

Definition 5.3 *(ElimTE).* The function *Elim*, which takes an extended arrow type as input and returns a set of arrow types, is defined by:

$$\begin{aligned}&\mathit{Elim}(q) = \{q\}\\ &\mathit{Elim}(\tau_1 \wedge \cdots \wedge \tau_m \to \tau) = \mathit{Elim}(\tau_1 \wedge \cdots \wedge \tau_m \wedge \alpha \to \tau) =\\ &\qquad \{\theta_1 \wedge \cdots \wedge \theta_m \to \theta \mid \theta_i \in \mathit{Elim}'(\tau_i), \theta \in \mathit{Elim}(\tau)\}\\ &\mathit{Elim}'(\tau) = \begin{cases} \mathit{Elim}(\tau) \cup \{\top\} & \text{if } \tau \text{ contains a type variable}\\ \mathit{Elim}(\tau) & \text{otherwise}\end{cases}\end{aligned}$$

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

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

In the definition above, the auxiliary function Elim' may return a set consisting of arrow types and a special type $\top$. We treat $\top$ as the unit on $\wedge$, i.e. we identify $\bigwedge\{\tau_1, \ldots, \tau_n, \top\} \to \tau$ with $\bigwedge\{\tau_1, \ldots, \tau_n\} \to \tau$. In the definition of $\mathit{Elim}(\tau_1 \wedge \cdots \wedge \tau_m \to \tau)$, 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:

$$\begin{aligned}&\mathit{Elim}((q_0 \wedge q_1 \to q_2) \wedge (q_0 \wedge \alpha \to q_2) \to q)\\ &= \{\theta_1 \wedge \theta_2 \to q \mid \theta_1 \in \mathit{Elim}'(q_0 \wedge q_1 \to q_2), \theta_2 \in \mathit{Elim}'(q_0 \wedge \alpha \to q_2)\}\\ &= \{\theta_1 \wedge \theta_2 \to q \mid \theta_1 \in \{q_0 \wedge q_1 \to q_2\}, \theta_2 \in \{q_0 \to q_2, \top\}\}\\ &= \{(q_0 \wedge q_1 \to q_2) \to q, (q_0 \wedge q_1 \to q_2) \wedge (q_0 \to q_2) \to q\}\end{aligned}$$

Remark 5.1. 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 $\Gamma = \{F : (q_0 \wedge q_1 \to q_2) \wedge (q_0 \wedge \alpha \to q_2) \to q\}$. If we replace α with $\top$, we obtain the type environment $\{F : (q_0 \wedge q_1 \to q_2) \wedge (q_0 \to q_2) \to q\}$. However, the actual type of F required to type the recursion scheme may be $(q_0 \wedge q_1 \to q_2) \to q$.

EXAMPLE 5.4. Consider the configuration graph $\mathcal{C}$ of Figure 9, for the running example. We have:

$$\begin{array}{l}
\tau_{S,N_0} = q_0 \\
\tau_{F,N_1} = \bigwedge(\{\tau_{\mathtt{c},N_2}, \tau_{\mathtt{c},N_5}\} \cup \alpha_1) \to \tau_{F\,\mathtt{c},N_1} = (q_0 \wedge q_1 \wedge \alpha) \to q_0 \\
\tau_{F,N_3} = \bigwedge(\{\tau_{\mathtt{b}\,\mathtt{c},N_4}\} \cup \alpha_2) \to \tau_{F\,(\mathtt{b}\,\mathtt{c}),N_3} = (q_0 \wedge \alpha_2) \to q_0 \\
\tau_{F,N_6} = \alpha_3 \to \tau_{F\,(\mathtt{b}\,(\mathtt{b}\,\mathtt{c})),N_6} = \alpha_3 \to q_0
\end{array}$$

Here, the type variables $\alpha_1, \alpha_2, \alpha_3$ denote the types of $\mathtt{c}\ \mathtt{b}\ \mathtt{c}$, and $\mathtt{b}\ (\mathtt{b}\ \mathtt{c})$, respectively, in the open node N_6.

$\Gamma_{\mathcal{C}}$ and $\mathit{ElimTE}(\Gamma_{\mathcal{C}})$ are given by:

$$\begin{array}{rcl}
\Gamma_{\mathcal{C}} & = & \{S : q_0, F : (q_0 \wedge q_1 \wedge \alpha_1) \to q_0, F : (q_0 \wedge \alpha_2) \to q_0, F : \alpha_3 \to q_0\} \\
\mathit{ElimTE}(\Gamma_{\mathcal{C}}) & = & \{S : q_0, F : q_0 \wedge q_1 \to q_0, F : q_0 \to q_0, , F : \top \to q_0\}.
\end{array}$$

□

5.3 Step 3

Step 3 takes a type environment Γ as an input, and checks whether there exists a subset Γ' of Γ such that $\Gamma' \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$. For that purpose, we use Lemma 4.11. The first line of Step 3 in Figure 8 computes the largest $\Gamma' \subseteq \Gamma$ such that $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma'$. Then, $S : q_{\mathcal{B},0} \in \Gamma'$ is checked on the second line.

EXAMPLE 5.5. Recall the result of Step 2 (Example 5.4) for the running example in Figure 6:

$$\Gamma = \{S : q_0, F : q_0 \wedge q_1 \to q_0, F : q_0 \to q_0, F : \top \to q_0\}.$$

As $\mathcal{F}^1_{\mathcal{G},\mathcal{B}}(\Gamma) = \mathcal{F}^2_{\mathcal{G},\mathcal{B}}(\Gamma) = \{S{:}q_0, F{:}q_0 \wedge q_1 \to q_0\}$, we obtain $\Gamma_1 = \{S{:}q_0, F{:}q_0 \wedge q_1 \to q_0\}$ as a fixed-point of $\mathcal{F}_{\mathcal{G},\mathcal{B}}$. Since $S{:}q_0 \in \Gamma_1$, the algorithm terminates and outputs Γ_1. □

5.4 Correctness of the Algorithm

This section proves the correctness of the algorithm:

THEOREM 5.1. *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 $\mathcal{B}^\perp$). If the algorithm reports an error path, then $[\![\mathcal{G}]\!]$ is not accepted by $\mathcal{B}^\perp$.*

The most difficult part is to show that the algorithm eventually terminates when $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^\perp$. To prove it, we first show that if a *closed* (possibly infinite) configuration graph is given, we can extract enough type information from it.

THEOREM 5.2. *Suppose that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}$. If $\mathcal{C}$ is a closed (possibly infinite) configuration graph of $\mathcal{G}$ over $\mathcal{B}$, then $\mathcal{G}$ is well-typed under $\Gamma_{\mathcal{C}}$, i.e., $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$.*

The theorem above can also be deduced from the proof of the completeness of Kobayashi and Ong's type system [Kobayashi and Ong 2009].[7] We give a direct proof in Appendix B.

By the theorem above, it suffices to show that from a sufficiently large, finitely expanded graph $\mathcal{C}'$, we can extract all the necessary type information, i.e., $\mathit{ElimTE}(\Gamma_{\mathcal{C}'}) \supseteq \Gamma_{\mathcal{C}}$ holds for the closed configuration graph $\mathcal{C}$ (c.f. Lemma 5.5 below).

We prepare a few definitions and lemmas before proving the property mentioned above. 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.

Definition 5.4. Let $\mathcal{C}$ be a closed configuration graph and $\mathcal{C}'$ a finitely expanded graph of a recursion scheme. Suppose that the node $\mathcal{C}(\pi)$ is labeled by $[t\,\widetilde{s}, q]$. The relation $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$ is defined by induction on the structure of the sort of t, as follows.

(1) If t has sort o, and $\tau_{t,\mathcal{C}'(\pi)} = q$, then $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$.
(2) Suppose t has sort $\kappa_1 \to \kappa_2$. Then, $\widetilde{s}$ must be of the form $s_0\widetilde{s'}$, and $\tau_{t,\mathcal{C}(\pi)}$ is of the form $\tau_1 \wedge \cdots \wedge \tau_m \to \tau_{ts_0,\mathcal{C}(\pi)}$. $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$ holds if the following conditions are satisfied:
   (i) $\mathcal{C}' \preceq_{\pi,t\,s_0} \mathcal{C}$;
   (ii) 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 enough information about the type $\tau_{t,\mathcal{C}(\pi)}$, as stated by the following lemma.

Lemma 5.3. *Let $\mathcal{C}$ be the closed configuration graph. If $\mathcal{C}' \preceq_{\pi,t} \mathcal{C}$, then the following conditions hold.*

(1) If $\mathcal{C}''$ is obtained from expansions of $\mathcal{C}'$, then $\mathcal{C}'' \preceq_{\pi,t} \mathcal{C}$.
(2) $\tau_{t,\mathcal{C}(\pi)} \in \mathit{Elim}(\tau_{t,\mathcal{C}'(\pi)})$.

Proof. Let the label of $\mathcal{C}(\pi)$ be $[t\,\widetilde{s}, q]$. The proof proceeds by induction on the sort of t. If the sort of t is 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) $\tau_{t,\mathcal{C}(\pi)} = \tau_1 \wedge \cdots \wedge \tau_m \to \tau$;
(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 (1), we have $\mathcal{C}'' \preceq_{\pi,ts_0} \mathcal{C}$ and $\mathcal{C}'' \preceq_{\pi\pi_i,s_0} \mathcal{C}$, which imply (1). By the induction hypothesis (2), we also have $\tau_{ts_0,\mathcal{C}(\pi)} \in \mathit{Elim}(\tau_{ts_0,\mathcal{C}'(\pi)})$ and

[7] More precisely, Kobayashi and Ong [Kobayashi and Ong 2009] considered a run tree corresponding to the closed configuration graph, gave the definition of $\tau_{F,N}$, and proved a theorem corresponding to Theorem 5.2. They did not consider how to extract type information from a finitely expanded configuration graph.

$\tau_{s_0,\mathcal{C}(\pi\pi_i)} \in \mathit{Elim}(\tau_{s_0,\mathcal{C}'(\pi\pi_i)})$. By the definition of $\tau_{t,\mathcal{C}'(\pi)}$, it is of the form:

$$\tau_{s_0,\mathcal{C}'(\pi\pi_1)} \wedge \cdots \wedge \tau_{s_0,\mathcal{C}'(\pi\pi_m)} \wedge \tau'_1 \wedge \cdots \wedge \tau'_k \to \tau_{ts_0,\mathcal{C}'(\pi)}.$$

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 $\mathit{Elim}'(\tau'_j)$, choose $\top$ when τ'_j contains a type variable; otherwise choose τ'_j (in which case we have $\tau'_j \in \{\tau_1, \ldots, \tau_m\}$).

Thus, we have $\tau_{t,\mathcal{C}(\pi)} \in \mathit{Elim}(\tau_{t,\mathcal{C}'(\pi)})$ as required. □

The next lemma says that for every node of the closed configuration graph, the corresponding node of a sufficiently large, finitely expanded graph carries enough type information.

LEMMA 5.4. *Let $N = \mathcal{C}(\pi)$ be a node of a closed configuration graph $\mathcal{C}$, and suppose that N is labeled with $[t\,\widetilde{s}, q]$. 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 `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 $\tau_1 \wedge \cdots \wedge \tau_m \to \tau$, 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 by Lemma 5.3 (1). (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.) □

Now we are ready to prove the key lemma.

LEMMA 5.5. *Suppose that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^\perp$, and let $\mathcal{C}$ be the closed configuration graph for $\mathcal{G}$ and $\mathcal{B}$. Then, there exists a finitely expanded graph $\mathcal{C}'$ such that $\Gamma_{\mathcal{C}} \subseteq \mathit{ElimTE}(\Gamma_{\mathcal{C}''})$ for every finite expansion $\mathcal{C}''$ of $\mathcal{C}'$.*

PROOF. For each $F_i{:}\tau_{i,j} \in \Gamma_{\mathcal{C}}$, pick a node $N_{i,j} = \mathcal{C}(\pi_{i,j})$ such that $\tau_{F_i,N_{i,j}} = \tau_{i,j}$. By Lemma 5.4, there exists a finitely expanded graph $\mathcal{C}_{i,j}$ such that $\mathcal{C}_{i,j} \preceq_{\pi_{i,j},F_i} \mathcal{C}$. Let $\mathcal{C}'$ be the union of all such $\mathcal{C}_{i,j}$'s, and $\mathcal{C}''$ be a finite expansion of $\mathcal{C}'$. By Lemma 5.3 (1), $\mathcal{C}'' \preceq_{\pi_{i,j},F_i} \mathcal{C}$ for every i, j. Thus, by Lemma 5.3 (2), we have $\Gamma_{\mathcal{C}} \subseteq \mathit{ElimTE}(\mathcal{C}'')$ as required. □

We now prove the main theorem.

PROOF OF THEOREM 5.1. If the algorithm terminates, then the soundness of the output follows immediately from the definition of Step 1 and Lemma 4.11. Thus, it remains to show that the algorithm eventually terminates.

Suppose that $[\![\mathcal{G}]\!]$ is not accepted by $\mathcal{B}^{\perp}$. Then an error path is eventually found, by the construction of the configuration graph in Step 1 and the fairness assumption on the selection of nodes.

Suppose that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}^{\perp}$, but that the algorithm does not terminate. Let $\mathcal{C}_i$ be the configuration graph constructed by the i-th iteration of Step 1, and $\mathcal{C}$ the union of all $\mathcal{C}_i$'s. Then, by the fairness of node selection, $\mathcal{C}$ must be a closed configuration graph. By Lemma 5.5, there exists i such that $\Gamma_{\mathcal{C}} \subseteq \mathit{ElimTE}(\Gamma_{\mathcal{C}_i})$. By Lemma 4.11, Step 3 generates the *largest* type environment Γ such that $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ and $\Gamma \subseteq \mathit{ElimTE}(\Gamma_{\mathcal{C}_i})$. By Theorem 5.2, $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma_{\mathcal{C}}$ and $\Gamma_{\mathcal{C}} \subseteq \mathit{ElimTE}(\Gamma_{\mathcal{C}_i})$ with $S : q_{\mathcal{B},0} \in \Gamma_{\mathcal{C}}$. Thus, we have $S : q_{\mathcal{B},0} \in \Gamma_{\mathcal{C}} \subseteq \Gamma$, which implies that the algorithm must terminate at the i-th iteration, hence a contradiction.

□

5.5 Discussion

This section discusses some properties, optimizations and limitations of the hybrid algorithm.

5.5.1 *A Minimality Property.* A main difference between the naive algorithm in Section 4.3 and the hybrid algorithm is that the former computes the set of all the valid types of non-terminals, while the latter tries to find, in a demand-driven manner, a smaller set of valid types that are just sufficient for typing the given recursion scheme. We show that the certificate found by the hybrid algorithm is indeed minimal in a certain sense.

Suppose that $\mathcal{B}$ accepts the tree generated by a recursion scheme $\mathcal{G}$. We define a binary relation $\sqsubseteq$ [Tsukada and Kobayashi 2010] on types inductively by:[8]

$$q \sqsubseteq q \qquad \frac{\theta \sqsubseteq \theta' \qquad \forall i \in \{1,\ldots,k\}.\exists j \in \{1,\ldots,\ell\}.\theta_i \sqsubseteq \theta'_j}{\bigwedge\{\theta_1,\ldots,\theta_k\} \to \theta \sqsubseteq \bigwedge\{\theta'_1,\ldots,\theta'_\ell\} \to \theta'}$$

For example, $q_1 \to q_0 \sqsubseteq q_1 \wedge q_2 \to q_0$ and $(q_1 \to q_0) \to q_0 \sqsubseteq (q_1 \wedge q_2 \to q_0) \wedge (q_3 \to q_1) \to q_0$ hold. Note that $\sqsubseteq$ is not a subtype relation: function types are covariant in argument types. The relation is extended to the binary relation on type environments by: $\Gamma \sqsubseteq \Gamma' \iff \forall (x:\theta) \in \Gamma.\exists (x:\theta') \in \Gamma'.\theta \sqsubseteq \theta'$. Note that $\sqsubseteq$ is a preorder (i.e. satisfies the transitivity and reflexivity). By definition, $\Gamma \subseteq \Gamma'$ implies $\Gamma \sqsubseteq \Gamma'$, but not vice versa.

We show that if the hybrid algorithm outputs Γ_h, then $\Gamma_h \sqsubseteq \Gamma$ holds for every Γ such that $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$; in other words, the hybrid algorithm outputs a minimal valid type environment with respect to the partial order induced by $\sqsubseteq$ (i.e. the quotient of $\sqsubseteq$ by $(\sqsubseteq \cup \sqsupseteq)^*$).

First, by Lemma 5 of Tsukada and Kobayashi [2010], the type environment obtained from the closed configuration graph is a minimal valid type environment:

Lemma 5.6 [Tsukada and Kobayashi 2010]. *Suppose* $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$. *If* $\mathcal{C}$ *is a closed configuration graph, then* $\Gamma_{\mathcal{C}} \sqsubseteq \Gamma$.

Suppose that Γ_h is obtained from a configuration graph $\mathcal{C}'$. Then by the definition of Step 3, $\Gamma_h \subseteq \mathit{ElimTE}(\Gamma_{\mathcal{C}'})$. By the construction of $\mathit{ElimTE}(\Gamma_{\mathcal{C}'})$, we have

[8] Tsukada and Kobayashi [2010] defined the relation co-inductively, as they considered infinite trees as types.

$\mathit{ElimTE}(\Gamma_{\mathcal{C}'}) \sqsubseteq \Gamma_{\mathcal{C}}$. (Note that $\Gamma_{\mathcal{C}}$ is obtained from $\Gamma_{\mathcal{C}'}$ by replacing type variables with certain types.) Thus, we have $\Gamma_h \sqsubseteq \mathit{ElimTE}(\Gamma_{\mathcal{C}'}) \sqsubseteq \Gamma_{\mathcal{C}}$. By using the lemma, we have $\Gamma_h \sqsubseteq \Gamma$ for any Γ such that $\Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$. Furthermore, since $\Gamma_h \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_{\mathcal{B},0}$, the above lemma also implies $\Gamma_{\mathcal{C}} \sqsubseteq \Gamma_h$. Thus, Γ_h is equivalent to $\Gamma_{\mathcal{C}}$ up to the equivalence relation $(\sqsubseteq \cup \sqsupseteq)^*$.

Although Γ_h is not necessarily the smallest with respect to the subset relation, the above result indicates that the hybrid algorithm is likely to output a much smaller type environment than the naive algorithm.

5.5.2 *Optimizations.* Here we discuss a few optimizations of the algorithm, which have been applied to the implementation of our model checker TRecS (see Section 6). The optimizations preserve the completeness of the algorithm. We only provide hints for proofs of the correctness of optimizations. More formal proofs are tedious but not difficult, and can be obtained by modifying the correctness proof of the unoptimized algorithm.

Optimization of Elim. The operation *Elim* used in Step 2 (Definition 5.3) may cause a combinatorial explosion of the number of types. The following optimizations reduce the number of arrow types without losing the completeness of the algorithm.

(1) Use canonical representation of function types. Here, a function type $\bigwedge_{i=1}^{n} \tau_i \to \tau$ is canonical if for each i, there is no $j \neq i$ such that $\tau_j \leq \tau_i$, where $\leq$ is the subtyping relation defined by:

$$q \leq q \qquad \frac{\theta \leq \theta' \wedge \forall i \in \{1,\ldots,m\}.\exists j \in \{1,\ldots,\ell\}.\theta'_j \leq \theta_i}{\bigwedge\{\theta_1,\ldots,\theta_m\} \to \theta \leq \bigwedge\{\theta'_1,\ldots,\theta'_\ell\} \to \theta'}$$

Accordingly, we need to extend the type system used in Step 3 with the subsumption rule:

$$\frac{\Gamma \vdash_{\mathcal{B}} t : \theta' \qquad \theta' \leq \theta}{\Gamma \vdash_{\mathcal{B}} t : \theta}$$

The correctness of this optimization follows from the soundness of the type system extended with subtyping, and the fact that if $\tau \in \mathit{Elim}(\tau_{t,N})$ then there exists $\tau' \in \mathit{Elim}_{\mathtt{opt1}}(\tau_{t,N})$ such that $\tau' \leq \tau$ and $\tau \leq \tau'$, where $\mathit{Elim}_{\mathtt{opt1}}$ is the optimized version of the function *Elim*. Ong and Ramsay [2009] discuss this optimization in more detail and prove the correctness.

(2) In the definition of $\mathit{Elim}(\tau_1 \wedge \cdots \wedge \tau_m \to \tau)$, choose $\tau'_i = \top$ from $\mathit{Elim}'(\tau_i)$ only if τ_i is subsumed by τ_j for some j, i.e. if there is a substitution ρ such that $\rho\tau_i \in \mathit{Elim}(\tau_j)$ for some j. The correctness of this optimization follows from the fact that Lemma 5.3 remains valid after the optimization. Note that in the construction of $\tau_{t,\mathcal{C}(\pi)}$ in the proof of Lemma 5.3, each τ'_j must be subsumed by some τ_i, which is in turn subsumed by $\tau_{s_0,C'(\pi\pi_i)}$. Thus, $\top$ can be chosen from $\mathit{Elim}'_{\mathtt{opt2}}(\tau'_j)$ where $\mathit{Elim}'_{\mathtt{opt2}}$ is the optimized version of Elim'.

Example 5.6. Let τ_1 be:

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

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

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

consisting of 2^3 types. With the first optimization, $\mathit{Elim}(\tau_1)$ generates:

$$\{(q_0 \wedge q_1 \to q_2) \to q, (q_1 \to q_2) \to q, (q_0 \to q_2) \to q,\\ (q_1 \to q_2) \wedge (q_0 \to q_2) \to q, (\top \to q_2) \to q\},$$

which consists of 5 types.

Let τ_2 be:

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

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

$$\{\bigwedge S \to q \mid S \subseteq \{q_0 \to q_2, q_1 \to q_2\}\},$$

which consists of 4 types. With the second optimization, $\mathit{Elim}(\tau)$ generates a singleton set:

$$\{(q_0 \to q_2) \wedge (q_1 \to q_2) \to q\}.$$

□

Pruning Nodes by Using Type Information. In Step 1, we can suppress the expansion of a node labeled with a term that is known to be well-typed. Let Γ be the type environment output by Step 3 of the previous iteration. Suppose that a node N is labeled with $\langle t, q, \mathtt{open}\rangle$ and that there is a type derivation tree Π for $\Gamma \vdash_{\mathcal{B}} t : q$. Then, we need not reduce t, intuitively because we already have enough type information about t. Instead of reducing t, for each node $\Gamma \vdash_{\mathcal{B}} s_i : \tau_i$ of the derivation tree Π, add special (closed) nodes N_i labeled with $[s_i, \tau_i]$ to the configuration graph, and add edges from N to those nodes. In Step 2, define τ_{s_i,N_i} to be τ_i if N_i is labeled with $[s_i, \tau_i]$. The rest of the algorithm remains the same.

To observe the correctness of this optimization, it suffices to see that Theorem 5.2 can be modified to:

> *Suppose that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{B}$ and that $\vdash_{\mathcal{B}} \mathcal{G} : \Gamma$ holds. If $\mathcal{C}$ is an (extended version of) closed configuration graph of $\mathcal{G}$ constructed by using $\mathcal{G}$, then $\Gamma_{\mathcal{C}} \cup \Gamma \vdash_{\mathcal{B}} (\mathcal{G}, S)$;*

and that the other properties used in the proof of Theorem 5.1 (in particular, Lemma 5.3) remain valid.

Example 5.7. Consider the following recursion scheme $\mathcal{G}_3$:

$$S \to \mathtt{br}\ (F\ \mathtt{a}\ \mathtt{c})\ (F\ \mathtt{b}\ \mathtt{c}) \qquad F\ f\ x \to \mathtt{br}\ (f\ x)\ (F\ f\ (f\ x))$$

The lefthand side of Figure 11 shows a configuration graph for the recursion scheme $\mathcal{G}_3$ and the automaton $\mathcal{B}_0$ in Example 2.3. There are two open nodes, labeled by $(F\ \mathtt{a}\ (\mathtt{a}\ \mathtt{c}), q_0)$ and $(F\ \mathtt{b}\ \mathtt{c}, q_0)$. By running Steps 2 and 3 for this configuration graph, we obtain a type environment $\Gamma = F : (q_0 \to q_0) \to q_0 \to q_0$.

Using the type environment, the term $F\ \mathtt{a}\ (\mathtt{a}\ \mathtt{c})$ is typed as follows.

$$\dfrac{\Gamma \vdash_{\mathcal{B}} F : (q_0 \to q_0) \to q_0 \to q_0 \quad \Gamma \vdash_{\mathcal{B}} \mathtt{a} : q_0 \to q_0 \qquad \dfrac{\Gamma \vdash_{\mathcal{B}} \mathtt{a} : q_0 \to q_0 \quad \Gamma \vdash_{\mathcal{B}} \mathtt{c} : q_0}{\Gamma \vdash_{\mathcal{B}} \mathtt{a}\ \mathtt{c} : q_0}}{\Gamma \vdash_{\mathcal{B}} F\ \mathtt{a}\ (\mathtt{a}\ \mathtt{c}) : q_0}$$

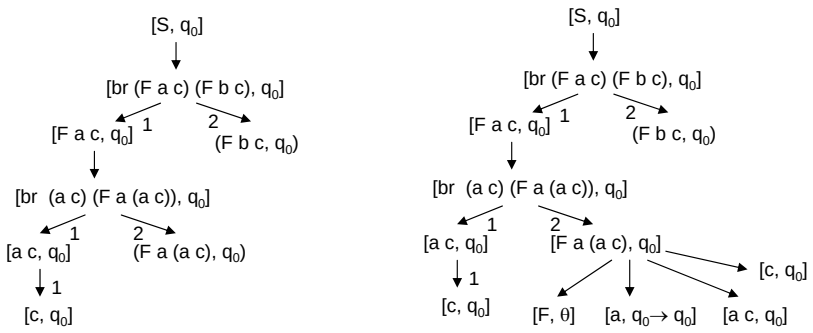


Fig. 11. Configuration Graphs Before/After Pruning ($\theta = (q_0 \to q_0) \to q_0 \to q_0$).

Thus, the node $(F\,\mathtt{a}\,(\mathtt{a}\ \mathtt{c}), q_0)$ can be pruned as shown on the righthand side of Figure 11.

□

5.5.3 *Limitations.* The main limitation of the algorithm described in Section 5 (especially from a theoretical point of view) is that the worst-case time complexity is not optimal. According to the result of Section 4.3, the time complexity of recursion scheme model checking (with respect to deterministic trivial automata) is linear in the size of recursion schemes, provided that the sizes of sorts and the trivial automaton are bounded above by a constant. The worst-case time complexity of the hybrid algorithm is, however, at least k-fold exponential under the same assumption. For example, consider the following order-1 recursion scheme for generating a word (or, a monadic tree) $\mathtt{a}^{2^m}\,\mathtt{c}$.

$$S \to F_0\,G,$$
$$F_0\,x \to F_1(F_1\,x), \quad F_1\,x \to F_2(F_2\,x), \quad \ldots, \quad F_{m-1}\,x \to F_m(F_m\,x), \quad F_m\,x \to \mathtt{a}\,x,$$
$$G \to \mathtt{c}$$

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, even without recursion, we can construct an order-k recursion scheme of size m for which $O(\mathbf{exp}_k(m))$ expansions of configuration graphs are required. Consider the following recursion scheme $\mathcal{G}_{k,m}$:

$$S \to F_0\ G_{k-1}\ \cdots\ G_2\ G_1\ G_0,$$
$$F_0\ f\ \widetilde{x} \to F_1\,(F_1\ f)\ \widetilde{x}, \quad \ldots, \quad F_{m-1}\ f\ \widetilde{x} \to F_m\,(F_m\ f)\ \widetilde{x}, \quad F_m\ f\ \widetilde{x} \to G_k\ f\ \widetilde{x},$$
$$G_k\ f\ z\ \widetilde{x} \to f\,(f\,z)\,\widetilde{x}, \quad \ldots, \quad G_2\ f\ z \to f\,(f\ z), \quad G_1\ z \to \mathtt{a}\ z, \quad G_0 \to \mathtt{c}$$

Here, F_i's have sort κ_k and G_j has sort κ_j, where κ_j is defined by:

$$\kappa_0 = \mathtt{o} \qquad \kappa_{j+1} = \kappa_j \to \kappa_j.$$

Note that the largest sort κ_k is independent of m. S is reduced to $\mathtt{a}^{\mathbf{exp}_k(m)}(G_0)$ and then to $\mathtt{a}^{\mathbf{exp}_n(m)}(\mathtt{c})$. (Notice that with η-conversion, S can be reduced as: $S \longrightarrow^* G_n^{2^m}\ G_{k-1}\ G_2\ G_1\ G_0 \longrightarrow^* G_{n-1}^{2^{2^m}}\ G_{k-2}\ G_2\ G_1\ G_0 \longrightarrow^* \cdots \longrightarrow^* G_1^{\mathbf{exp}_k(m)}\ G_0$.) Thus,

our algorithm requires $O(\mathbf{exp}_k(m))$ expansions to extract the type information about G_0.

Remark 5.2. λ-terms corresponding to $\mathcal{G}_{k,m}$ have been previously used to show lower bounds of various problems about λ-calculus [Henglein and Mairson 1994; Mairson 1992; Beckmann 2001].

Let us discuss an upper bound of the minimum number of expansions required by our algorithm. Beckmann [2001] showed that the length of a reduction sequence of a typed λ-term of order-k is bounded by $\mathbf{exp}_k(m)$, where m is the size of the term. Using this result, we can obtain an upper bound $\mathbf{exp}_{k+1}(|\mathcal{G}|)$ for order-k recursion schemes *without recursion*. (The reason for the exponential blow-up with respect to the bound for λ-terms is that an order-$(k+1)$ λ-term is required for representing an order-k recursion scheme without recursion; for instance, the recursion scheme above is represented by $(\lambda F_0.F_0\cdots)((\lambda F_1.\lambda f.\lambda\widetilde{x}.F_1(F_1\ f)\,\widetilde{x})\cdots)$, where the subterm $\lambda F_0.F_0\cdots$ has order $k+1$.)

For general recursion schemes, an upper bound of $O(\mathbf{exp}_{k+1}(|\mathcal{G}|^2 \times \mathbf{exp}_k((A|Q|)^{1+\epsilon})))$ is obtained as follows. Let m be $|\Gamma_{\mathbf{max}}|+1$, which is $O(|\mathcal{G}| \times \mathbf{exp}_k((A\times|Q|)^{1+\epsilon}))$ (recall Section 4.3). Let $\mathcal{G}^{(m)}$ be the (recursion-free) recursion scheme constructed in the proof of Theorem 4.6. Let $\mathcal{C}$ be the closed configuration graph for $\mathcal{G}^{(m)}$. By the strong normalization of the simply-typed λ-calculus, $\mathcal{C}$ is finite and obtained in a finite number of expansions. By Theorem 5.2, $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}^\perp} (\mathcal{G}^{(m)}, S^{(m)}) : q_0$. Let $\Gamma_\ell(\ell \in \{0,\ldots,m\})$ be: $\{F:\theta \mid F^{(j)}:\theta \in \Gamma_{\mathcal{C}}, j \geq \ell\}$. Then, we have:

$$\Gamma_m \subseteq \Gamma_{m-1} \subseteq \cdots \subseteq \Gamma_1 \subseteq \Gamma_0.$$

As $\Gamma_0 \subseteq \Gamma_{\mathbf{max}}$ and $m > |\Gamma_{\mathbf{max}}|$, there must exist j such that $\Gamma_j = \Gamma_{j+1}$. By the condition $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} (\mathcal{G}^{(m)}, S^{(m)}) : q_0$, we have $\Gamma_j \vdash_{\mathcal{B}} (\mathcal{G}, S) : q_0$.

Now, let $\mathcal{C}'$ be the configuration graph of $\mathcal{G}$ corresponding to $\mathcal{C}$, obtained by expanding exactly the nodes whose corresponding nodes in $\mathcal{C}$ are expanded. Then, $\Gamma_j \subseteq \mathit{ElimTE}(\Gamma_{\mathcal{C}'})$, hence the algorithm terminates at this point. The number of expansions to obtain $\mathcal{C}'$ is the same as the one to obtain $\mathcal{C}$, which is bounded by $O(\mathbf{exp}_{k+1}(|\mathcal{G}^{(m)}|)) = O(\mathbf{exp}_{k+1}(|\mathcal{G}|^2 \times \mathbf{exp}_k((A|Q|)^{1+\epsilon})))$ by using the result of Beckmann [2001]. Obtaining a tighter upper bound is left for future work.

6. EXPERIMENTS

We have implemented a model checker TRecS (Types for RECursion Scheme, `http://www-kb.is.s.u-tokyo.ac.jp/~koba/trecs/`) for recursion schemes. To our knowledge, this is the first implementation of a recursion scheme model checker.

The implementation is based on the algorithm described in Section 5. Following are some details.

- In Step 1, nodes are expanded in a depth-first manner, except that the expansions of nodes corresponding to deeply nested recursive function calls are delayed.
- In Step 3, Rehof and Mogensen's algorithm [Rehof and Mogensen 1999] is used for the fixed-point computation.
- All the optimizations discussed in Section 5.5.2 are applied. In particular, before expanding a node (t,q) in Step 2, it is checked whether t is well typed under the type environment found so far.

Recursion schemes	O	R	S	Q	result	E	time
`Example 2.1`	1	2	8	2	YES	90	1
`Example 2.2`	2	3	11	2	YES	301	2
`Example 3.1`	4	7	27	4	YES	20	1
`Example 3.3`	4	7	25	4	NO	7	1
`Example 3.5`	4	10	36	1	YES	88	2
`Example 3.6`	3	6	18	3	YES	13	1
`Section 3.3.3`	3	10	31	1	YES	10	1
`Example 3.7`	3	9	48	2	YES	96	2
`Example 5.2`	1	2	8	2	NO	9	1
`Twofiles`	4	11	47	5	YES	67	1
`TwofilesWrong`	4	11	45	5	NO	38	1
`TwofilesExn`	4	12	56	5	YES	75	2
`FileOcamlc`	4	23	111	4	YES	200	3
`Lock1`	4	12	46	3	YES	48	2
`Lock2`	4	11	45	4	YES	313	9
`Order5`	5	11	52	5	YES	58	1
`Order5-2`	5	9	40	5	YES	117	6

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

We have tested the model checker for a number of hand-written and machine-generated recursion schemes. The experiment reported below was conducted on an (unloaded) machine with an Intel(R) Xeon(R) CPU with 3Ghz and 8GB memory. The times shown in the tables are the maximum running times for 5 runs.

Table I shows the result for several hand-written recursion schemes. Most of them were obtained from program verification problems, based on the reductions discussed in Section 3. 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). The column "E" shows the number of expansions of the configuration graph. The column "time" shows the running time, measured in milliseconds. In this experiment, we have set the number of iterations in Step 1 (the value of `MAX` in Figure 8) to 200.

The first 10 recursion schemes were taken from earlier sections of this paper, indicated by the names of the programs. For Examples 2.1 and 2.2, the specification is given by the automaton in Example 2.3.

The rest of the programs were obtained from resource usage verification problems by using the reduction discussed in Section 3. `Twofiles` is the recursion scheme generated from the program `twofiles` discussed in Section 3.2. `TwofilesWrong` is the recursion scheme obtained from a wrong variant of `Twofiles`, where `close(x)` has been removed.

The program `FileOcamlc` is based on a part of the source code of Objective Caml compiler 3.11.0 [Oca], `bytecomp/symtable.ml`, which is the most interesting and complex use of input files we found in the compiler source code. 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 a record consisting of closures for accessing the file recursively. The main body obtains such a record, and uses it inside recursion. This kind of program could not be handled by previous (incomplete) methods for the resource usage verification [Igarashi and Kobayashi 2005; Iwama et al. 2006]. The original source code consists of about 60 lines of code. We obtained the recursion scheme from it by manually slicing irrelevant parts, applying CPS transformation, and then applying the reduction described in Section 3.2.

The recursion scheme `Lock2` has been obtained from the following program, where the resource usage specification is that every lock should be used according to $(\texttt{lock} \cdot \texttt{unlock})^*$.

```
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 tricky, as (i) infinitely many locks are created, and (ii) locks are stored in closures and accessed through them.

`Order5` is an order-5 recursion scheme obtained from 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 "foo" in let genw () = open_out "bar" in
   gencon genr read close; gencon genw write close
```

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

The other examples are explained in Appendix C.

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 model checking of order-k recursion schemes is $(k-1)$-EXPTIME in general. Note that all the previous model checking algorithms [Ong 2006; Kobayashi 2009b; Kobayashi and Ong 2009] are simply non-executable for the recursion schemes of orders 4 or 5, because of huge requirement for time and space (recall the discussion in Section 5.3).

Following 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`, model checking timed out when we used pure breadth-first search.
- Without the optimizations described in Section 5.2, the model checker ran out of stack space in Step 2 for the programs "FileOcamlc" and "Lock2". That is due to the combinatorial explosion of the number of atomic types, introduced by *Elim* (recall Example 5.6). After the optimizations, however, the number of atomic types is kept small.

Table II shows the result of experiments for larger recursion schemes, automatically generated from program verification tools [Kobayashi et al. 2010; Kobayashi et al. 2011] constructed on top of TRecS. In the experiments, the parameter `MAX` was set to 400. The column "time" again shows the running time, measured in milliseconds. The recursion schemes `repeat` and `mc91` were generated from the following functional programs.

```
let rec repeat f n s = if n=0 then s else f(repeat f (n-1) s) in
let succ x = x+1 in assert (repeat succ n 0 >= 0)

let rec mc91 x = if x>100 then x-100 else mc91(mc91(x+11))
in if n<=101 then assert(mc91 n = 91)
```

The goal of the verification is to check that an assertion failure never occurs. The programs above have been transformed to recursion scheme model checking by a combination of techniques for predicate abstraction and the reduction of reachability problem (recall Section 3.3.1): see [Kobayashi et al. 2011] for more details. Note that the order of the recursion scheme `repeat` is very high. The type of `repeat` in the source program is $(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$, but the order of the program is raised by CPS transformation to translate a call-by-value program to a recursion scheme.

The recursion schemes `gapid`, ..., `xhtmlf-m` were generated from verification problems for XML processing programs [Tozawa 2006; Frisch and Hosoya 2007], based on the reduction described by [Kobayashi et al. 2010]. The source programs for the recursion schemes `xhtmlf-*` take an XHTML document as input, and output another XHTML document; the goal of the verification is to check that the output is a valid XHTML document. Although the order of recursion schemes is not high, the number of states is large.

The verification tools mentioned above actually generate recursion schemes extended with finite data domains (such as booleans) [Kobayashi et al. 2010]. For the experiments above, we have encoded values of finite data domains into functions by using Church encoding: the value a_i of a finite data domain $\{a_1, \ldots, a_n\}$ is represented by $\lambda x_1, \ldots, x_n.x_i$. The actual verification tools [Kobayashi et al. 2010; Kobayashi et al. 2011] use an extension of TRecS that directly supports finite data domains, which is several times faster for `xhtmlf-id` and `xhtmlf-m`.

To confirm the limitation discussed in Section 5.5.3, we have also tested TRecS for model checking of the recursion scheme $\mathcal{G}_{k,m}$ given in Section 5.5.3. The checked property is that the tree contains only terminals `a` and `c`, which holds trivially. Table III shows the running time for the model checking of $\mathcal{G}_{k,m}$. In the experiments,

Table II. Result for machine-generated recursion schemes.

Recursion schemes	O	R	S	Q	result	E	time (msec.)
`repeat`	8	40	191	1	YES	184	5
`mc91`	4	49	358	1	YES	998	54
`gapid`	3	24	182	9	YES	154	8
`xhtmlf-id`	2	52	2889	50	YES	800	1,800
`xhtmlf-div`	2	64	3003	50	NO	75	51
`xhtmlf-m`	2	64	3027	50	YES	1099	894

Table III. Verification time for $\mathcal{G}_{k,m}$ (in seconds).

	m=1	m=2	m=3	m=4	m=5	m=10	m=15
k=1	0.002	0.002	0.002	0.002	0.003	0.036	2.866
k=2	0.002	0.002	0.011	228.412	–	–	–
k=3	0.002	394.267	–	–	–	–	–

the parameter `MAX` was set to $20,000$. The cell marked by "–" indicates that the model checker did not terminate in 10 minutes. As expected, the model checking of $\mathcal{G}_{k,m}$ rapidly slows down with an increase of k.

7. RELATED WORK

Model checking of higher-order recursion schemes. Higher-order grammars, where non-terminals can take functions as parameters, have been introduced in early 70s [Turner 1972; Wand 1974], and actively studied in 80s [Damm 1982]. To our knowledge, it is Knapik et al. [2001; 2002] who first formalized the modal μ-calculus model checking problems for higher-order recursion schemes in the present form. Knapik et al. [2001] 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 recursion schemes of any order [Knapik et al. 2002]. Their proof reduces a model checking problem for an order-k safe recursion scheme to that for an order-$(k-1)$ safe recursion scheme. Knapik et al. [2005] and Aehlig et al. [2005] then independently showed that the model checking problem is decidable for order-2 recursion schemes, without the safety assumption. Finally, Ong [2006] has shown that the problem is decidable for recursion schemes of arbitrary order. He used game semantics to reduce the model checking problem to parity games over variable profiles. Hague et al. [2008] and Kobayashi and Ong [2009] later gave alternative proofs of the decidability of the modal μ-calculus model checking of recursion schemes. The former [Hague et al. 2008] used the equi-expressivity between higher-order recursion schemes and collapsible higher-order pushdown automata, and reduced the model checking problem to a parity game over the configuration graph of a collapsible pushdown automaton. The latter [Kobayashi and Ong 2009] extended the type system in Section 4 and reduced the model checking problem to a type checking problem in the extended type system. For the class of trivial automata, Aehlig [2007] gave a simpler decidability proof of recursion scheme model checking, based on a set-theoretic interpretation of tree functions. Our type system is similar to his approach; in fact, the semantics of our intersection types

can be given in terms of set-theoretic functions. All the decidability proofs above are constructive in the sense that they all provide algorithms for model checking recursion schemes. As already mentioned, however, all the algorithms almost always suffer from k-EXPTIME bottleneck. As a natural consequence, there has been no implementation of a model checker for recursion schemes.

Kobayashi [2011] recently proposed yet another practical algorithm for recursion scheme model checking. Unlike the hybrid algorithm, his new algorithm runs in time linear in the size of higher-order recursion schemes if the largest size of types and the size of the automaton are fixed. For $\mathcal{G}_{k,m}$ discussed in Section 5.5.3, the new algorithm runs much faster than the hybrid algorithm. For the recursion schemes obtained from program verification algorithms, however, the current implementation of the new algorithm [Kobayashi 2011] is significantly slower than the hybrid algorithm. Thus, further investigation is necessary to obtain a better (fixed-parameter) linear time algorithm.

The complexity of the modal μ-calculus model checking of order-k recursion schemes was shown to be k-EXPTIME complete by Ong [2006]. His proof of the lower bound is based on Cachat and Walukiewicz's result on k-EXPTIME hardness of the reachability game on a higher-order pushdown system [Cachat and Walukiewicz 2007]. As mentioned in Remark 2.1, Kobayashi and Ong [2011] studied the complexity of model checking of recursion schemes for subclasses of the modal μ-calculus.

Despite the theoretical interests in recursion scheme model checking, there has been little work on its application to program verification, probably due to the extremely high worst-case complexity. To our knowledge, Kobayashi [2009b] was the first to show concrete applications of recursion scheme model checking to verification of higher-order functional programs. Succeedingly, recursion scheme model checking has been applied to various verification or analysis problems of functional programs [Kobayashi et al. 2010; Ong and Ramsay 2011; Kobayashi et al. 2011; Tobita et al. 2012].

Software model checking. Model checking has been recently applied to software verification [Ball et al. 2001; Ball and Rajamani 2002; Beyer et al. 2007; Henzinger et al. 2002]. Dillig et al. [2008] has also developed a closely related technique for sound and complete path-sensitive analysis that is scalable to million lines of code. We are not, however, 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 [Ball et al. 2001; Beyer et al. 2007] 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 [Ball et al. 2001] deals with function pointers by replacing them with non-deterministic jumps to the functions they may point to.

Bakewell and Ghica [2008] 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.

Many of the techniques for model checking, especially techniques for predicate

abstraction and CEGAR [Graf and Saïdi 1997; Ball et al. 2001; Clarke et al. 2003; Henzinger et al. 2002; McMillan 2006], are useful also in the context of higher-order model checking, and some of them have been applied already [Kobayashi et al. 2011].

Relationship between model checking and type systems. Naik and Palsberg [2003; 2005] studied type systems equivalent to model checking 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 [2003; 2005] use union types in addition to intersection types. The combination of union and intersection types would be useful also in the context of recursion scheme model checking, to enable more compact representation of types. For example, the intersection type $\bigwedge\{\theta_1 \to \theta_2 \to \mathsf{o} \mid \theta_1, \theta_2 \in \{q_0, q_1\}\}$ can be compactly represented as $(q_0 \vee q_1) \to (q_0 \vee q_1) \to \mathsf{o}$.

Type-based program analysis. Type systems have been a popular technique for program analysis and verification [Palsberg 2001]. Various type systems for the resource usage verification problem considered in Section 3.1 or its variants have been proposed [Igarashi and Kobayashi 2005; Foster et al. 2002; DeLine and Fähndrich 2001; Iwama et al. 2006]. Unlike our method based on higher-order model checking, those type systems are not complete, and some of them [DeLine and Fähndrich 2001] require type annotations.

Connections between types and tree automata have been studied in the context of languages for XML processing [Hosoya et al. 2005; Benzaken et al. 2003]. They deal with *finite* trees, while our type system deals with infinite trees. Type annotations for recursive functions are required in their languages.

Intersection types and their variants have been used for various program analyses [Jensen 1991; Henglein and Mossin 1994; Freeman and Pfenning 1991; Mossin 2003]. Mossin [2003] developed an intersection type system for *exact* flow analysis, which provides a sound and complete algorithm for flow analysis of the simply-typed λ-calculus with recursion and finite base types. Like previous model checking algorithms for recursion schemes, his algorithm is inefficient and mainly of theoretical interest. Jensen [1992] also gave an intersection type system that is equivalent to Burn, Hankin, and Abramsky's strictness analysis for higher-order programs [Burn et al. 1986]. Freeman and Pfenning [1991] introduced refinement types as a refinement of ML type system to infer precise properties of ML data types. Our development of the type system and the naive type inference algorithm in Section 4 is similar to this line of work; A key principle shared by this line of research is that there are only finitely many intersection types that refine each standard type, so that one can use a fixed-point computation to infer intersection types, as in our naive algorithm discussed in Section 4.3. The most important difference of our work from those previous studies is invention of the hybrid type inference algorithm. The algorithms used in previous studies (and our naive algorithm) are based on a fixed-point computation. They are not feasible for higher-order programs, as they try to

enumerate all the valid types. Instead, our hybrid algorithm tries to infer minimal types just enough to type a given program, so that it does not immediately suffer from the explosion of the number of refinements. This difference is analogous, in the context of ML type inference, to the difference between the standard ML type inference algorithm W, which tries to assign the most general type scheme to each function, and Bjørner's algorithm to find minimal typing derivations, which tries to assign the most specific type to each function (though the motivation is different).

Another difference from previous studies on type-based program analysis is that our type system for recursion scheme model checking can be used for a general purpose, in the sense that various problems can be reduced to model checking problems, which can be solved by using our type system.

Inference of Intersection Types. Since our model checking algorithm is a type inference algorithm for the intersection type system presented in Section 4, there may be some connection between our algorithm and type inference algorithms for intersection types [Coppo et al. 1980; Ronchi Della Rocca and Venneri 1984; Kfoury and Wells 2004; Boudol 2008]. In particular, earlier algorithms for intersection type inference [Coppo et al. 1980; Ronchi Della Rocca and Venneri 1984] first find a normal form, and then obtain a principal typing for the normal form; this is somewhat similar to Steps 1 and 2 of our hybrid algorithm, which first reduces a given recursion scheme, and then extracts 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 [Coppo et al. 1980; Ronchi Della Rocca and Venneri 1984; Kfoury and Wells 2004; Boudol 2008] guarantee that typable terms (possibly with certain additional conditions) have the strong normalization property, while our type system 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.

8. CONCLUSION

We have proposed a novel framework for program verification based on model checking of higher-order recursion schemes. We have also developed new algorithms for recursion scheme model checking, and implemented the first practical model checker for recursion schemes.

It is too early to judge how much impact our program verification framework has in practice. There are, however, a number of reasons to believe that our approach is promising and worthy of further investigation:

- As already mentioned, our verification method is sound, *complete*, and *fully automatic* for the simply-typed λ-calculus with recursion and finite base types.

- Higher-order model checking can be considered an extension of finite state and pushdown model checking, which have already achieved certain success in software verification [Beyer et al. 2007; Ball and Rajamani 2002].
- The model checking algorithm is actually a type checking algorithm, so that it can take the best of model checking and type-based analysis. In particular, as in other type-based approach, our algorithm generates type information as a certificate of successful verification, and as in model checking, the algorithm generates an error trace as a counterexample.
- Our verification method can be integrated with testing. Recall that our hybrid algorithm gathers type information by actually running a program (or reducing a recursion scheme). Such type information can be collected during testing.

Because of the extremely high worst-case complexity of recursion scheme model checking, obtaining a more scalable model checker remains a major challenge. There is however a good reason to hope that recursion scheme model checking may scale for realistic inputs, if we come up with a proper model checking algorithm. According to the discussion of the worst-case behavior of our hybrid algorithm in Section 5.5.3, the algorithm can be very slow in two cases. One is the case where we have to unfold recursive definitions very deeply to extract enough type information. The other is the case where, even without recursion, a term has a very long reduction sequence. The first case does not happen if higher-order functions are used in a uniform manner so that they have only a small number of different types. The second case happens because of the ability of higher-order functions to express a long computation very compactly, which should actually be regarded as an advantage of higher-order functions, rather than as a limitation of higher-order model checking. For example, consider an order-0 recursion scheme $\mathcal{G}_0$:

$$S \to H_0, \quad H_0 \to \mathtt{a}\, H_1, \quad H_1 \to \mathtt{a}\, H_2, \quad \cdots \quad H_{2^m-1} \to \mathtt{a}\, H_{2^m}, \quad H_{2^m} \to \mathtt{c},$$

and an order-1 recursion scheme $\mathcal{G}_1$:

$$S \to F_0(\mathtt{c}), \quad F_1\ x \to F_2(F_2\ x), \quad \cdots \quad F_{m-1}\ x \to F_m(F_m\ x), \quad F_m\ x \to \mathtt{a}\ x.$$

The two recursion schemes generate the same tree $\mathtt{a}^{2^m}(\mathtt{c})$, but the former is exponentially larger than the latter. Thus, an exponential algorithm for the latter recursion scheme is in fact as good as a polynomial time algorithm for the former recursion scheme. Furthermore, the naive model checking algorithm in Section 4.3 and the algorithm in [Kobayashi 2011] run in time linear in the size of recursion schemes, so that the model checking of the latter recursion scheme can be performed exponentially faster than the model checking of the former recursion scheme. This indicates that our verification method can be faster for well-structured programs that use higher-order functions in an appropriate manner, than for unstructured programs that do not use higher-order functions.

We conclude this article with discussion of future work.

- More efficient implementation of a recursion scheme model checker. The current implementation of TRecS is reasonably fast considering the worst-case complexity, but it will not scale to verification of very large programs. It would be interesting to investigate a combination of the hybrid algorithm and the fixed-

parameter linear time algorithm recently proposed in [Kobayashi 2011]. BDD-like implementation techniques would be important for the scalability.

- Dealing with a larger class of programs. Higher-order model checking is sound and complete only for the *simply-typed* λ-calculus with recursion and *finite* base types. Although predicate abstraction techniques can be used for dealing with infinite data domains such as integers, it remains a challenge to deal with other features supported in programming languages, such as recursive types, polymorphism, references, and concurrency.
- Compositional verification. In Section 3, we have discussed only verification of closed programs, where all the definitions of functions are given at the time of verification. To enable compositional verification, we need to extend recursion schemes so that the start symbol of a recursion scheme can take parameters $f_1, \ldots, f_k$ (which represent unknown functions), and need to solve questions like "assuming $f_1, \ldots, f_k$ satisfy certain conditions, does the tree generated by $S\, f_1 \ldots, f_k$ satisfy a given condition?" and "what are sufficient conditions on $f_1, \ldots, f_k$, in order for the tree generated by $S\, f_1 \ldots, f_k$ to satisfy a given condition?"

  Fortunately, our type-based model checking method seems suitable for such extensions. If the conditions of parameters are expressed by using types, the naive type checking algorithm can be used to answer both types of questions.[9] The hybrid algorithm can also be used to answer the first type of question, and in combination with testing, it can also be used to partially answer the second type of question.
- Dealing with a larger class of properties. In this article, we have restricted our attention to the class of properties expressed by deterministic trivial automata. Based on the result of Kobayashi and Ong [2009], it is not difficult to extend our hybrid algorithm to deal with the full modal μ-calculus model checking. Implementing an efficient model checker that supports the full modal μ-calculus remains a challenge, however. On the theoretical side, it would be a challenge to identify a class of non-regular properties (such as counting properties) for which recursion scheme model checking is decidable. At the moment, we have only a negative result [Kobayashi 2010] on this issue.

Acknowledgment

We would like to thank anonymous referees for a number of useful comments. We would also like to thank Luke Ong, who introduced higher-order recursion schemes to me and gave a number of insightful comments. We have also benefited from working with him on related topics. We are also grateful to Ryosuke Sato, Naoshi Tabuchi, and Hiroshi Unno for providing machine-generated recursion schemes for the experiments in Section 6, and many other people for useful comments and discussion, including Fritz Henglein, Atsushi Igarashi, Jens Palsberg, Sriram Rajamani, Eijiro Sumii, Tachio Terauchi, and Takeshi Tsukada. This work was partially supported by Kakenhi 20240001 and 23220001.

[9] The algorithm is not complete, however: Because of the undecidability of λ-definability [Loader 2001], we cannot decide whether there exists a parameter that has a given type.

REFERENCES

Objective caml. `http://caml.inria.fr/ocaml/`.

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

Aehlig, K., de Miranda, J. G., and Ong, C.-H. L. 2005. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In *TLCA 2005*. Lecture Notes in Computer Science, vol. 3461. Springer, 39–54.

Appel, A. W. 1992. *Compiling with Continuations*. Cambridge University Press.

Bakewell, A. and Ghica, D. R. 2008. 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*. Lecture Notes in Computer Science, vol. 4963. Springer, 78–92.

Ball, T., Majumdar, R., Millstein, T. D., and Rajamani, S. K. 2001. Automatic predicate abstraction of C programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 203–213.

Ball, T. and Rajamani, S. K. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1–3.

Barendregt, H., Coppo, M., and Dezani-Ciancaglini, M. 1983. A filter lambda model and the completeness of type assignment. *J. Symb. Log. 48,* 4, 931–940.

Beckmann, A. 2001. Exact bounds for lengths of reductions in typed lambda-calculus. *J. Symb. Log. 66,* 3, 1277–1285.

Benzaken, V., Castagna, G., and Frisch, A. 2003. CDuce: an XML-centric general-purpose language. In *Proceedings of ICFP 2003*. ACM Press, 51–63.

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

Blume, M., Acar, U. A., and Chae, W. 2008. Exception handlers as extensible cases. In *Proceedings of APLAS 2008*. Lecture Notes in Computer Science, vol. 5356. Springer, 273–289.

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

Burn, G. L., Hankin, C., and Abramsky, S. 1986. Strictness analysis for higher-order functions. *Sci. Comput. Program. 7,* 3, 249–278.

Cachat, T. and Walukiewicz, I. 2007. The complexity of games on higher order pushdown automata. *CoRR abs/0705.0262*.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the Association for Computing Machinery (JACM) 50,* 5, 752–794.

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

Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., and Vardi, M. Y. 2007. Proving that programs eventually do something good. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 265–276.

Coppo, M., Dezani-Ciancaglini, M., and Sallé, P. 1979. Functional characterization of some semantic equalities inside lambda-calculus. In *Proceedings of ICALP 1979*. Lecture Notes in Computer Science, vol. 71. Springer, 133–146.

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

Coppo, M., Dezani-Ciancaglini, M., and Venneri, B. 1981. Functional characters of solvable terms. *Mathematical Logic Quarterly 27,* 2-6, 45–58.

Courcelle, B. 1983. Fundamental properties of infinite trees. *Theoretical Computer Science 25*, 95–169.

Damas, L. and Milner, R. 1982. Principal type-schemes for functional programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 207–212.

Damas, L. M. M. 1984. Type assignment in programming languages. Ph.D. thesis, University of Edinburgh.

Damm, W. 1982. The IO- and OI-hierarchies. *Theoretical Computer Science 20*, 95–207.

Danvy, O. and Filinski, A. 1992. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science 2,* 4, 361–391.

DeLine, R. and Fähndrich, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 59–69.

Dillig, I., Dillig, T., and Aiken, A. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of PLDI 2008*. ACM Press, 270–280.

Emerson, E. A. and Jutla, C. S. 1991. Tree automata, mu-calculus and determinacy (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science (FOCS 1991)*. IEEE Computer Society Press, 368–377.

Foster, J. S., Terauchi, T., and Aiken, A. 2002. Flow-sensitive type qualifiers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 1–12.

Freeman, T. and Pfenning, F. 1991. Refinement types for ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 268–277.

Frisch, A. and Hosoya, H. 2007. Towards practical typechecking for macro tree transducers. In *Database Programming Languages, 11th International Symposium (DBPL 2007)*. Lecture Notes in Computer Science, vol. 4797. Springer, 246–260.

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *Proceedings of CAV 97*. Lecture Notes in Computer Science, vol. 1254. Springer, 72–83.

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

Henglein, F. and Mairson, H. G. 1994. The complexity of type inference for higher-order typed lambda calculi. *Journal of Functional Programming 4,* 4, 435–477.

Henglein, F. and Mossin, C. 1994. Polymorphic binding-time analysis. In *Proceedings of 5th European Symposium on Programming (ESOP'94)*. Lecture Notes in Computer Science, vol. 788. Springer, 287–301.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. 2002. Lazy abstraction. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 58–70.

Hosoya, H., Vouillon, J., and Pierce, B. C. 2005. Regular expression types for XML. *ACM Trans. Program. Lang. Syst. 27,* 1, 46–90.

Igarashi, A. and Kobayashi, N. 2005. Resource usage analysis. *ACM Transactions on Programming Languages and Systems 27,* 2, 264–313.

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

Jensen, T. P. 1991. Strictness analysis in logical form. In *Functional Programming Languages and Computer Architecture (FPCA 91)*. Lecture Notes in Computer Science, vol. 523. Springer, 352–366.

Jensen, T. P. 1992. Abstract interpretation in logical form. Ph.D. thesis, Imperial College.

Johnsson, T. 1985. Lambda lifting: Treansforming programs to recursive equations. In *Proceedings of FPCA 85*. Lecture Notes in Computer Science, vol. 201. Springer, 190–203.

Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. 1990. ML typability is DEXTIME-complete. In *Proceedings of CAAP '90*. Lecture Notes in Computer Science, vol. 431. Springer, 206–220.

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

Knapik, T., Niwinski, D., and Urzyczyn, P. 2001. Deciding monadic theories of hyperalgebraic trees. In *TLCA 2001*. Lecture Notes in Computer Science, vol. 2044. Springer, 253–267.

Knapik, T., Niwinski, D., and Urzyczyn, P. 2002. Higher-order pushdown trees are easy. In *FoSSaCS 2002*. Lecture Notes in Computer Science, vol. 2303. Springer, 205–222.

Knapik, T., Niwinski, D., Urzyczyn, P., and Walukiewicz, I. 2005. Unsafe grammars and panic automata. In *ICALP 2005*. Lecture Notes in Computer Science, vol. 3580. Springer, 1450–1461.

Knaster, B. 1927. Un théorème sur les fonctions d'ensembles. *Ann. Soc. Polon. Math. 6*, 133–134.

Kobayashi, N. 2009a. Model-checking higher-order functions. In *Proceedings of PPDP 2009*. ACM Press, 25–36.

Kobayashi, N. 2009b. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 416–428.

Kobayashi, N. 2010. Balancedness of high-level word languages is undecidable. draft.

Kobayashi, N. 2011. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *Proceedings of FoSSaCS 2011*. Lecture Notes in Computer Science, vol. 6604. Springer, 260–274.

Kobayashi, N. and Ong, C.-H. L. 2009. 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, 179–188.

Kobayashi, N. and Ong, C.-H. L. 2011. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science 7,* 4.

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

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

Kupferman, O., Vardi, M. Y., and Wolper, P. 2000. An automata-theoretic approach to branching-time model checking. *Journal of the Association for Computing Machinery (JACM) 47,* 2, 312–360.

Leroy, X. and Pessaux, F. 2000. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems 22,* 2, 340–377.

Loader, R. 2001. Finitary PCF is not decidable. *Theoretical Computer Science 266,* 1-2, 341–364.

Mairson, H. G. 1990. Deciding ML typability is complete for deterministic exponential time. In *POPL*. ACM Press, 382–401.

Mairson, H. G. 1992. A simple proof of a theorem of statman. *Theoretical Computer Science 103,* 2, 387–394.

McMillan, K. L. 2006. Lazy abstraction with interpolants. In *Proceedings of CAV 2006*. Lecture Notes in Computer Science, vol. 4144. Springer, 123–136.

Meyer, A. R. and Wand, M. 1985. Continuation semantics in typed lambda-calculi (summary). In *Logic of Programs*. Lecture Notes in Computer Science, vol. 193. Springer, 219–224.

Might, M. and Shivers, O. 2008. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming 18,* 5-6, 821–864.

Mossin, C. 2003. Exact flow analysis. *Mathematical Structures in Computer Science 13,* 1, 125–156.

Mycroft, A. 1980. The theory and practice of transforming call-by-need into call-by-value. In *Symposium on Programming*. Lecture Notes in Computer Science, vol. 83. Springer, 269–281.

Naik, M. 2003. A type system equivalent to a model checker. Master Thesis, Purdue University.

Naik, M. and Palsberg, J. 2005. A type system equivalent to a model checker. In *ESOP 2005*. Lecture Notes in Computer Science, vol. 3444. Springer, 374–388.

Nielson, F., Nielson, H. R., and Hankin, C. 1999. *Principles of Program Analysis*. Springer.

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

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

Ong, C.-H. L. and Ramsay, S. J. 2009. Subtyping for model checking recursion schemes. Preprint, July 6, 2009. A summary appeared in Proceedings of the Oxford University Computing Laboratory Student Conference 2009, CS-RR-09-14.

Palsberg, J. 2001. Type-based analysis and applications. In *Proceedings of PASTE'01*. ACM Press, 20–27.

Plotkin, G. D. 1975. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci. 1,* 2, 125–159.

Rehof, J. and Mogensen, T. 1999. Tractable constraints in finite semilattices. *Science of Computer Programming 35,* 2, 191–221.

Ronchi Della Rocca, S. and Venneri, B. 1984. Principal type schemes for an extended type theory. *Theor. Comput. Sci. 28*, 151–169.

Schwoon, S. 2002. Model-checking pushdown systems. Ph.D. thesis, Technische Universität München.

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

Tobita, Y., Tsukada, T., and Kobayashi, N. 2012. Exact flow analysis by higher-order model checking. In *Proceedings of FLOPS 2012*. Lecture Notes in Computer Science, vol. 7294. Springer, 275–289.

Tozawa, A. 2006. XML type checking using high-level tree transducer. In *Functional and Logic Programming, 8th International Symposium (FLOPS 2006)*. Lecture Notes in Computer Science, vol. 3945. Springer, 81–96.

Tsukada, T. and Kobayashi, N. 2010. Untyped recursion schemes and infinite intersection types. In *Proceedings of FOSSACS 2010*. Lecture Notes in Computer Science, vol. 6014. Springer, 343–357.

Turner, R. 1972. An infinite hierarchy of term languages - an approach to mathematical complexity. In *Proceedings of ICALP*. North-Holland, 593–608.

van Bakel, S. 1992. Complete restrictions of the intersection type discipline. *Theor. Comput. Sci. 102,* 1, 135–163.

van Bakel, S. 1995. Intersection type assignment systems. *Theor. Comput. Sci. 151,* 2, 385–435.

Wand, M. 1974. An algebraic formulation of the chomsky hierarchy. In *Category Theory Applied to Computation and Control*. Lecture Notes in Computer Science, vol. 25. Springer, 209–213.

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1, 38–94.

Yi, K. 1994. Compile-time detection of uncaught exceptions in Standard ML programs. In *Static Analysis, First International Static Analysis Symposium (SAS'94)*. Lecture Notes in Computer Science, vol. 864. Springer, 238–254.

Appendix

A. PROOF OF THEOREM 3.3

This section gives a proof of Theorem 3.3. We fix a program D and a resource automaton $W = (L, Q, \delta_W, q_0, Q_F)$ below (hence $\mathcal{G}_{D,W}$ and $\mathcal{B}_W = (\Sigma, Q \cup \{q_{\mathbf{untracked}}, q_{\mathbf{any}}\}, \delta', q_{\mathbf{untracked}})$ is also fixed).

The idea of the proof is to match a reduction sequence of the program D with a run of the automaton $\mathcal{B}_W$ over a path of the tree generated by $\mathcal{G}_{D,W}$. For that purpose, we introduce alternative reduction semantics for programs and recursion schemes.

An *extended run-time state* is a triple (H, e, R), where H is a finite map from variables to $Q \cup \{q_{\mathbf{untracked}}\}$, e is an expression, and R is either the empty set $\emptyset$ or a singleton set $\{x\}$. Intuitively, R is the set of resources that are currently tracked. An extended reduction relation $(H, e, R) \rightsquigarrow_{D,W} C$, where C is either a triple (H', e', R') or **Error**, is defined by:

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

$$(H, \mathbf{if\_}\ e_1\ e_2, R) \rightsquigarrow_{D,W} (H, e_1, R)$$

$$(H, \mathbf{if\_}\ e_1\ e_2, R) \rightsquigarrow_{D,W} (H, e_2, R)$$

$$(H, \mathbf{new}^q\ e, R) \rightsquigarrow_{D,W} (H\{x \mapsto q_{\mathbf{untracked}}\}, e\ x, R) \qquad (x \notin dom(H))$$

$$(H, \mathbf{new}^q\ e, \emptyset) \rightsquigarrow_{D,W} (H\{x \mapsto q\}, e\ x, \{x\}) \qquad (x \notin dom(H))$$

$$(H\{x \mapsto q\}, \mathbf{acc}_a\ x\ e, R) \rightsquigarrow_{D,W} (H\{x \mapsto \delta'_W(q,a)\}, e, R) \quad (\text{if } x \notin dom(H))$$

$$(H\{x \mapsto q\}, \mathbf{acc}_a\ x\ e, R) \rightsquigarrow_{D,W} \mathbf{Error} \quad (\text{if } x \notin dom(H) \text{ and } \delta'_W(q,a) \text{ is undefined})$$

$$(H\{x \mapsto q\}, \diamond, R) \rightsquigarrow_{D,W} \mathbf{Error} \qquad (\text{if } x \notin dom(H) \text{ and } q \notin Q_F \cup \{q_{\mathbf{untracked}}\})$$

Here, $\delta'_W = \delta_W \cup \{(q_{\mathbf{untracked}}, a) \mapsto q_{\mathbf{untracked}} \mid a \in L\}$. The main change from the original reduction rules is in the rules for $\mathbf{new}^q\ e$. When R is empty, it is nondeterministically decided whether the fresh resource x should be tracked (i.e. put into R). Untracked resources have the special state $q_{\mathbf{untracked}}$, and all following accesses to them are ignored.

The alternative semantics is equivalent to the original semantics in the sense that a program is reduced to **Error** in the former if and only if it is so in the latter.

LEMMA A.1. *Let D be a (well-typed) program and W be a resource automaton for D. Then, $(\emptyset, S) \longrightarrow^*_{D,W} \mathbf{Error}$ if and only if $(\emptyset, S, \emptyset) \rightsquigarrow^*_{D,W} \mathbf{Error}$.*

PROOF. Suppose $(\emptyset, S) \longrightarrow^*_{D,W} \mathbf{Error}$. Then the reduction sequence must be of the form

$$\begin{aligned}(\emptyset, S) &\longrightarrow^*_{D,W} (H_1, \mathbf{new}^q\ e_1) \longrightarrow_{D,W} (H_1\{x \mapsto q\}, e_1\, x) \\ &\longrightarrow^*_{D,W} (H_2, e_2) \longrightarrow_{D,W} \mathbf{Error},\end{aligned}$$

where either $e_2 = \mathbf{acc}_a\ x\ e_3$ with $\delta_W(H(x), a)$ being undefined, or $e_2 = \diamond$ with $H(x) \notin Q_F$. In either case, one can construct (by straightforward induction on the length of the reduction sequence) a corresponding reduction sequence:

$$(\emptyset, S, \emptyset) \rightsquigarrow^*_{D,W} (H_1, \mathbf{new}^q\ e_1, \emptyset) \rightsquigarrow_{D,W} (H_1\{x \mapsto q\}, e_1\, x, \{x\}) \rightsquigarrow^*_{D,W} (H_2, e_2, \{x\}) \rightsquigarrow_{D,W} \mathbf{Error}.$$

The converse is similar. □

We now introduce an alternative reduction semantics for recursion schemes. Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}_\mathcal{G}, S)$ be a recursion scheme and $\mathcal{B} = (\Sigma, Q, \delta_\mathcal{B}, q_0)$ be a trivial automaton. The relation $(q, t) \triangleright_{\mathcal{G},\mathcal{B}} C$, where C is of the form (q', t') or **Error**, is the least relation closed under the following rules.

$$(q, F\,\widetilde{u}) \triangleright_{\mathcal{G},\mathcal{B}} (q, [\widetilde{u}/\widetilde{x}]t) \text{ (if } \mathcal{R}_\mathcal{G}(F) = \lambda\widetilde{x}.t)$$

$$(q, a\,\widetilde{u}) \triangleright_{\mathcal{G},\mathcal{B}} (q_i, u_i) \text{ (if } \delta_\mathcal{B}(q, a) = q_1 \cdots q_m \text{ with } 1 \le i \le m)$$

$$(q, a\,\widetilde{u}) \triangleright_{\mathcal{G},\mathcal{B}} \mathbf{Error} \text{ (if } \delta_\mathcal{B}(q, a) \text{ is undefined)}$$

The reduction relation $\triangleright_{\mathcal{G},\mathcal{B}}$ expresses the process of running the automaton $\mathcal{B}$ along a path while expanding a term lazily.

Using the new semantics, the acceptance of the tree generated by a recursion scheme is characterized as follows.

LEMMA A.2. *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}_\mathcal{G}, S)$ be a recursion scheme and $\mathcal{B} = (\Sigma, Q, \delta_\mathcal{B}, q_0)$ be a trivial automaton. Then, $\mathcal{B}^\perp$ does not accept $[\![\mathcal{G}]\!]$ if and only if $(q_0, S) \triangleright^*_{\mathcal{G},\mathcal{B}}$ **Error**.*

PROOF. We write $(q, t) \triangleright'_\mathcal{B} (q', t')$ if $(q, t) \triangleright_{\mathcal{G},\mathcal{B}} (q', t')$ is derived without using the first rule. We first note the following commutativity properties, which follow immediately from the definitions of the reduction relations.

(i) If $t \longrightarrow_\mathcal{G} t'$ and $(q, t') \triangleright_{\mathcal{G},\mathcal{B}} (q', t'')$, then $(q, t) \triangleright^*_{\mathcal{G},\mathcal{B}} (q', t'')$ or $(q, t) \triangleright_{\mathcal{G},\mathcal{B}} (q', u)$ with $u \longrightarrow_\mathcal{G} t''$ for some u.

(ii) If $(q, t) \triangleright'_\mathcal{B} (q', t')$ and $t' \longrightarrow_\mathcal{G} t''$, then $t \longrightarrow_\mathcal{G} u$ with $(q, u) \triangleright'_{\mathcal{G},\mathcal{B}} (q', t'')$ for some u.

If $[\![\mathcal{G}]\!]$ is rejected by $\mathcal{B}^\perp$, then there is a term t such that $S \longrightarrow^*_\mathcal{G} t$ and $t^\perp$ is rejected by $\mathcal{B}^\perp$. Since $t^\perp$ is rejected by $\mathcal{B}^\perp$, we have $(q_0, t) \triangleright'^*_{\mathcal{G},\mathcal{B}}$ **Error**. By the property (i) above and $S \longrightarrow^*_\mathcal{G} t$, we have $(q_0, S) \triangleright^*_{\mathcal{G},\mathcal{B}} (q, u)$ and $u \longrightarrow^*_\mathcal{G} u'$, with $(q, u') \triangleright_{\mathcal{G},\mathcal{B}}$ **Error**. We can assume without loss of generality that the reductions $u \longrightarrow^*_\mathcal{G} u'$ do not rewrite the root symbol. Thus, we have $(q, u) \triangleright_{\mathcal{G},\mathcal{B}}$ **Error**, which implies $(q_0, S) \triangleright^*_{\mathcal{G},\mathcal{B}}$ **Error** as required.

If $(q_0, S) \triangleright^*_{\mathcal{G},\mathcal{B}}$ **Error**, then by the property (ii) above, there exists t such that $S \longrightarrow^*_\mathcal{G} t$ and $(q, t) \triangleright'^*_\mathcal{B}$ **Error**. Thus, $t^\perp$ is rejected by $\mathcal{B}^\perp$, which implies that $[\![\mathcal{G}]\!]$ is also rejected by $\mathcal{B}^\perp$. □

We are now ready to relate the reduction of a program and that of a recursion scheme. We write $(H, e, R) \sim (q, t)$ if either of the following conditions holds.

(i) $H = \{\widetilde{y} \mapsto q_{\mathbf{untracked}}\}$ and $R = \emptyset$ with $q = q_{\mathbf{untracked}}$ and $t = [K/\widetilde{y}]e$.
(ii) $H = \{x \mapsto q, \widetilde{y} \mapsto q_{\mathbf{untracked}}\}$ and $R = \{x\}$ with $q \in Q$ and $t = [I/x, K/\widetilde{y}]e$.

Here, $\{\widetilde{y} \mapsto q\}$ and $[K/\widetilde{y}]$ are abbreviations of $\{y_1 \mapsto q, \ldots, y_n \mapsto q\}$ and $[K/y_1, \ldots, K/y_n]$, respectively.

The following is the key lemma to prove Theorem 3.3..

LEMMA A.3. *Suppose that* $(H, e, R) \sim (q, t)$ *holds.*

(1) If $(H, e, R) \rightsquigarrow^*_{D,W}$ **Error**, *then* $(q, t) \triangleright^*_{\mathcal{G}_{D,W}, \mathcal{B}_W}$ **Error**.
(2) If $(q, t) \triangleright^*_{\mathcal{B}_W}$ **Error**, *then* $(H, e, R) \rightsquigarrow^*_{D,W}$ **Error**.

PROOF. We often omit the subscripts of $\rightsquigarrow^*$ and $\triangleright^*$ below. δ_W is the transition function of W, and $\delta'_W = \delta_W \cup \{(q_{\mathbf{untracked}}, a) \mapsto q_{\mathbf{untracked}} \mid a \in L\}$.

(1) The proof proceeds by induction on the length of the reduction sequence $(H, e, R) \rightsquigarrow^*_{D,W}$ **Error**, with case analysis on the shape of e. We define a substitution $\rho_{R,H}$ by:

$$\rho_{R,H} = \begin{cases} [K/\widetilde{y}] & \text{if } R = \emptyset \text{ and } dom(H) = \{\widetilde{y}\} \\ [I/x, K/\widetilde{y}] & \text{if } R = \{x\} \text{ and } dom(H) = \{x, \widetilde{y}\} \end{cases}$$

(a) Case $e = \diamond$:
In this case, there exists x such that $x \in R$ with $H(x) \notin Q_F$. By the definition of $(H, e, R) \sim (q, t)$, we have $t = \diamond$ and $q = H(x)$. Thus, $(q, t) \triangleright_{\mathcal{G}_{D,W}, \mathcal{B}_W}$ **Error**.
(b) Case $e = F\ \widetilde{e}_1$ with $F\ \widetilde{z} = e_0 \in D$:
In this case, $(H, e, R) \rightsquigarrow_{D,W} (H, e', R) \rightsquigarrow^*_{D,W}$ **Error**, where $e' = [\widetilde{e}_1/\widetilde{z}]e_0$ By the condition $(H, e, R) \sim (q, t)$, we have $t = \rho_{R,H}(F\ \widetilde{e}_1)$. Let t' be $\rho_{R,H}e'$. Then, we have $(H, e', R) \sim (q, t')$. By the induction hypothesis, we have $(q, t') \triangleright^*$ **Error**, which implies $(q, t) \triangleright (q, t') \triangleright^*$ **Error**.
(c) Case $e = \mathbf{if}\_\ e_1\ e_2$: In this case, $(H, e, R) \rightsquigarrow (H, e_i, R) \rightsquigarrow^*$ **Error** for some $i \in \{1, 2\}$. By the condition $(H, e, R) \sim (q, t)$, we have $t = \mathbf{if}\_\ \rho_{R,H}e_1\ \rho_{R,H}e_2$. Let t_i be $\rho_{R,H}e_i$. Then, $(H, e_i, R) \sim (q, t_i)$ holds. By the induction hypothesis, we have $(q, t_i) \triangleright^*$ **Error**, which implies $(q, t) \triangleright (q, \mathtt{br}\ t_1\ t_2) \triangleright (q, t_i) \triangleright^*$ **Error**.
(d) Case $e = \mathbf{new}^{q'}\ e_1$:
In this case, the reduction sequence $(H, e, R) \rightsquigarrow^*_{D,W}$ **Error** must be of the form $(H, e, R) \rightsquigarrow (H', e', R') \rightsquigarrow^*$ **Error**, where $H' = H \cup \{z \mapsto q_1\}$, $e' = e_1\ z$, and (q_1, R') is either $(q_{\mathbf{untracked}}, R)$ or $(q', R \cup \{z\})$. By the condition $(H, e, R) \sim (q, t)$, we have $t = \mathbf{new}^{q'}\ \rho_{R,H}e_1$. If $(q_1, R') = (q_{\mathbf{untracked}}, R)$, then $(H', e', R') \sim (q, t')$ for $t' = (\rho_{R,H}e_1)K = \rho_{R',H'}e'$. By the induction hypothesis, we have $(q, t') \triangleright^*$ **Error**, which implies $(q, t) \triangleright (q, \mathtt{br}\ t'\ (\nu^{q'}\ (\rho_{R,H}e_1\ I))) \triangleright (q, t') \triangleright^*$ **Error**.
If $(q_1, R') = (q', R \cup \{z\})$, then it must be the case that $R = \emptyset$ and $q = q_{\mathbf{untracked}}$. We have $(H', e', R') \sim (q', t')$ for $t' = (\rho_{R,H}e_1)I = \rho_{R',H'}e'$. By the induction hypothesis, we have $(q', t') \triangleright^*$ **Error**, which implies $(q, t) \triangleright (q, \mathtt{br}\ (\rho_{R,H}e_1\ K)\ (\nu^{q'}\ t')) \triangleright (q, \nu^{q'}\ t') \triangleright (q', t') \triangleright^*$ **Error**.
(e) Case $e = \mathbf{acc}_a\ x\ e_1$:
In this case, by the condition $(H, e, R) \sim (q, t)$, we have $t =$

$\mathbf{acc}_a\ (\rho_{R,H}x)\ (\rho_{R,H}e_1)$. By the condition $(H,e,R) \rightsquigarrow^*_{D,W} \mathbf{Error}$, we have either of the following conditions.

(i) $H(x) = q$, $R = \{x\}$, and $\delta'_W(q,a)$ is undefined.

(ii) $(H,e,R) \rightsquigarrow^*_{D,W} \mathbf{Error}$ is of the form $(H,e,R) \rightsquigarrow (H',e_1,R) \rightsquigarrow^* \mathbf{Error}$, where $H = H_1 \cup \{x \mapsto q_1\}$ and $H' = H_1 \cup \{x \mapsto \delta'_W(q_1,a)\}$.

In the first case, $t = \mathbf{acc}_a\ I\ (\rho_{R,H}e_1)$, so that we have $(q,t) \triangleright (q, I\,a\,(\rho_{R,H}e_1)) \triangleright (q, a\,(\rho_{R,H}e_1)) \triangleright \mathbf{Error}$.

In the second case, we perform case analysis on whether $x \in R$. If $x \in R$, then $t = \mathbf{acc}_a\ I\ (\rho_{R,H}e_1)$ with $q_1 = q$. Let $t' = \rho_{R,H}e_1 = \rho_{R,H'}e_1$. Then, since $(H',e_1,R) \sim (\delta_W(q_1,a),t')$, we have $(\delta_W(q_1,a),t') \triangleright^* \mathbf{Error}$ by the induction hypothesis. Thus, we have $(q,t) \triangleright^* (q_1, a\,t') \triangleright^* \mathbf{Error}$ as required. If $x \notin R$, then $t = \mathbf{acc}_a\ K\ (\rho_{R,H}e_1)$. Let $t' = \rho_{R,H}e_1 = \rho_{R,H'}e_1$. Then, since $(H',e_1,R) \sim (q,t')$, we have $(q,t') \triangleright^* \mathbf{Error}$ by the induction hypothesis. Thus, we have $(q,t) \triangleright^* (q, K\,a\,t') \triangleright (q,t') \triangleright^* \mathbf{Error}$ as required.

(2) The proof proceeds by induction on the length of the reduction sequence $(q,t) \triangleright^*_{\mathcal{B}_W} \mathbf{Error}$, with case analysis on the shape of e.

(a) Case where e is $\diamond$:
By the condition $(H,e,R) \sim (q,t)$, t must be $\diamond$. By the condition $(q,t) \triangleright^* \mathbf{Error}$, it must be the case that $q \notin Q_F \cup \{q_{\mathbf{untracked}}\}$. By using $(H,e,R) \sim (q,t)$ again, we get $R = \{x\}$ and $H(x) = q$. Thus, we have $(H,e,R) = (H,\diamond,R) \rightsquigarrow \mathbf{Error}$ as required.

(b) Case where e is of the form $F\ \widetilde{e_1}$ with $F\ \widetilde{z} = e_0 \in D$. By the condition $(H,e,R) \sim (q,t)$, it must be the case that $t = \rho_{R,H}(F\ \widetilde{e_1})$. The reduction sequence $(q,t) \triangleright^* \mathbf{Error}$ must be of the form:

$$(q,t) \triangleright (q,t') \triangleright^* \mathbf{Error}$$

where $t' = [\rho_{R,H}\widetilde{e_1}/\widetilde{z}]e_0$. Let e'' be $[\widetilde{e_1}/\widetilde{z}]e_0$. Then, we have $(H,e'',R) \sim (q,t')$. By the induction hypothesis, we have $(H,e'',R) \rightsquigarrow^* \mathbf{Error}$, which implies $(H,e,R) \rightsquigarrow (H,e'',R) \rightsquigarrow^* \mathbf{Error}$.

(c) Case where e is of the form $\mathbf{if}\_\ e_1\ e_2$:
By the condition $(H,e,R) \sim (q,t)$, it must be the case that $t = \rho_{R,H}(\mathbf{if}\_\ e_1\ e_2)$. The reduction sequence $(q,t) \triangleright^* \mathbf{Error}$ must be of the form:

$$(q,t) \triangleright (q, \rho_{R,H}(\mathtt{br}\ e_1\ e_2)) \triangleright (q, \rho_{R,H}e_i) \triangleright^* \mathbf{Error}$$

where i is either 1 or 2.
Let e'' be e_i. Then, we have:

$$(H,e'',R) \sim (q, \rho_{R,H}e_i).$$

By the induction hypothesis, we have $(H,e'',R) \rightsquigarrow^* \mathbf{Error}$, which implies $(H,e,R) \rightsquigarrow (H,e'',R) \rightsquigarrow^* \mathbf{Error}$.

(d) Case where e is of the form $\mathbf{new}^q\ e_1$:
By the condition $(H,e,R) \sim (q,t)$, it must be the case that $t = \mathbf{new}^q\ \rho_{R,H}e_1$. $(q,t) \triangleright^* \mathbf{Error}$ must be either of the form:

$$\begin{aligned}(q,t) &\triangleright (q, \mathtt{br}\ (\rho_{R,H}e_1\ K)\ (\nu^{q'}\,(\rho_{R,H}e_1\ I)))\\ &\triangleright (q, \rho_{R,H}e_1\ K)\\ &\triangleright^* \mathbf{Error}\end{aligned}$$

or, of the form:

$$\begin{array}{l}(q,t) \triangleright (q, \mathtt{br}\ (\rho_{R,H} e_1\ K)\ (\nu^q(\rho_{R,H} e_1\ I))) \\ \quad \triangleright (q, \nu^{q'}(\rho_{R,H} e_1\ I)) \\ \quad \triangleright (q', \rho_{R,H} e_1\ I) \\ \quad \triangleright^* \mathbf{Error}\end{array}$$

In the former case, let H' be $H \cup \{z \mapsto q'\}$ and e' be $e_1 z$. Then, we have:

$$(H', e', R) \sim (q, \rho_{R,H} e_1\ K).$$

By the induction hypothesis, we have $(H', e', R) \rightsquigarrow^* \mathbf{Error}$, which implies $(H, e, R) \rightsquigarrow (H', e', R) \rightsquigarrow^* \mathbf{Error}$.
In the latter case, it must be the case that $q = q_{\mathbf{untracked}}$ with $R = \emptyset$. Let $H' = H \cup \{z \mapsto q_{\mathbf{untracked}}\}$, $e' = e_1 z$, and $R' = \{z\}$. Then, we have $(H', e', R') \sim (q', \rho_{R,H} e_1\ I)$. By the induction hypothesis, we have $(H', e', R') \rightsquigarrow^* \mathbf{Error}$, which implies $(H, e, R) \rightsquigarrow (H', e', R') \rightsquigarrow^* \mathbf{Error}$ as required.

(e) Case where e is of the form $\mathbf{acc}_a\ z\ e_1$:
By the condition $(H, e, R) \sim (q, t)$, it must be the case that $t = \mathbf{acc}_a\ (\rho_{R,H} z)\ (\rho_{R,H} e_1)$. Let $H = H_1 \cup \{z \mapsto q_1\}$.
The reduction sequence $(q, t) \triangleright^* \mathbf{Error}$ must be one of the following forms.
  (i) $(q,t) \triangleright (q, I\, a\, (\rho_{R,H} e_1)) \triangleright (q, a\, (\rho_{R,H} e_1)) \triangleright \mathbf{Error}$, where $\delta_W(q,a)$ is undefined.
  (ii) $(q,t) \triangleright (q, I\, a\, (\rho_{R,H} e_1)) \triangleright (q, a\, (\rho_{R,H} e_1)) \triangleright (\delta_W(q,a), \rho_{R,H} e_1) \triangleright^* \mathbf{Error}$.
  (iii) $(q,t) \triangleright (q, K\, a\, (\rho_{R,H} e_1)) \triangleright (q, \rho_{R,H} e_1) \triangleright^* \mathbf{Error}$.
In the first or second case, we have $R = \{z\}$ and $q = q_1$. In the first case $(H, e, R) \rightsquigarrow \mathbf{Error}$ follows immediately. In the second case, we have $(H_1 \cup \{z \mapsto \delta_W(q,a)\}, e_1, R) \sim (\delta_W(q,a), \rho_{R,H} e_1)$. Thus, by the induction hypothesis, we have $(H_1 \cup \{z \mapsto \delta_W(q,a)\}, e_1, R) \rightsquigarrow^* \mathbf{Error}$, which implies $(H, e, R) \rightsquigarrow^* \mathbf{Error}$ as required.
In the third case, we have $z \notin R$ and $q_1 = q_{\mathbf{untracked}}$. By the condition $(H_1 \cup \{z \mapsto \delta'_W(q_1,a)\}, e_1, R) \sim (q, \rho_{R,H_1} e_1) = (q, \rho_{R,H} e_1)$ and the induction hypothesis, we have $(H_1 \cup \{z \mapsto \delta'_W(q_1,a)\}, e_1, R) \rightsquigarrow^* \mathbf{Error}$, which implies $(H, e, R) \rightsquigarrow^* \mathbf{Error}$ as required.

□

Theorem 3.3 follows as an immediate corollary of the above lemmas.

Proof of Theorem 3.3. We show the contraposition.

D is not resource-safe, i.e., $(\emptyset, S) \longrightarrow^*_{D,W} \mathbf{Error}$
if and only if $(\emptyset, S, \emptyset) \longrightarrow^*_{D,W} \mathbf{Error}$ (by Lemma A.1),
if and only if $(q,t) \triangleright^*_{\mathcal{G}_{D,W}, \mathcal{B}_W} \mathbf{Error}$ (by Lemma A.3),
if and only if $[\![\mathcal{G}_{D,W}]\!]$ is not accepted by $\mathcal{B}_W$ (by Lemma A.2).

□

B. PROOF OF THEOREM 5.2

PROOF. As the configuration graph contains the root node $[S, q_0]$, we have $S{:}q_0 \in \Gamma_{\mathcal{C}}$. Thus, it remains to prove that $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} \mathcal{R}(F) : \theta$ for each $F : \theta \in \Gamma_{\mathcal{C}}$. In view of proving this, we first construct a type derivation tree $\Pi_{t,N}$ of $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} t : \tau_{t,N}$ for every term t that occurs in a head position in a node N of the configuration graph $\mathcal{C}$ (i.e. the node N is labeled by $\langle t\widetilde{s}, q, \texttt{closed}\rangle$). The construction of $\Pi_{t,N}$ proceeds by induction on the structure of t.

(a) Case where t is a non-terminal F: In this case, $\Pi_{t,N}$ is just an application of rule T-VAR:

$$\frac{}{\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} F : \tau_{F,N}}\ \text{T-VAR}$$

Note that $F : \tau_{F,N} \in \Gamma_{\mathcal{C}}$ holds by the construction of $\Gamma_{\mathcal{C}}$.

(b) Case where t is a terminal a: In this case, $\Pi_{t,N}$ is an application of rule T-CONST:

$$\frac{}{\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} a : \tau_{a,N}}\ \text{T-CONST}$$

By the construction of the configuration graph, the node N must be labeled by $[a\, t_1 \cdots t_m, q]$, and have outgoing edges to $N_1, \ldots, N_m$ labeled by $[t_1, q_1], \ldots, [t_m, q_m]$, with $\delta_{\mathcal{B}}(q, a) = q_1 \cdots q_m$. By the construction, $\tau_{a,N}$ must be $q_1 \to \cdots \to q_m \to q$. Thus, the type derivation above is valid.

(c) Case where t is an application $t_1 t_2$: By the induction hypothesis, we have a type derivation $\Pi_{t_1,N}$ for $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} t_1 : \tau_{t_1,N}$. By the construction, $\tau_{t_1,N}$ must be:

$$\bigwedge\{\tau_{t_2,N_i} \mid N_i \in S\} \to \tau_{t,},$$

where S is the set of nodes where t_2 occurs in a head position. Since $\{\tau_{t_2,N_i} \mid N_i \in S\}$ is a finite set (although S may be infinite), there exists a finite subset $S' = \{N'_1, \ldots, N'_k\}$ of S that satisfies: (i) $\{\tau_{t_2,N'_i} \mid N'_i \in S'\} = \{\tau_{t_2,N_i} \mid N_i \in S\}$, and (ii) $\tau_{t_2,N'_1}, \ldots, \tau_{t_2,N'_k}$ are distinct from each other. By the induction hypothesis, we have type derivations Π_{t_2,N'_i} of $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} t_2 : \tau_{t_2,N'_i}$, for each $i \in \{1, \ldots, k\}$. Now, let $\Pi_{t_1 t_2,N}$ be:

$$\frac{\Pi_{t_1,N} \quad \Pi_{t_2,N'_1} \quad \cdots \quad \Pi_{t_2,N'_k}}{\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} t_1 t_2 : \tau_{t_1 t_2,N}}\ \text{T-APP}$$

An important property of the type derivation tree constructed above is that every node of the derivation tree of $\Pi_{t,N}$ is labeled by $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} s : \tau_{s,N'}$, where s is a subterm of t and occurs in a head position of N'.

Now, suppose $F : \theta \in \Gamma_{\mathcal{C}}$ and $\mathcal{R}(F) = \lambda x_1, \ldots, x_m.s$. We need to show $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} \lambda x_1, \ldots, x_m.s : \theta$. By the construction of $\Gamma_{\mathcal{C}}$, it must be the case that $\theta = \tau_{F,N}$ for some node N of $\mathcal{C}$. The node N must be labeled with $[F\, t_1 \cdots t_m, q]$, and have a single outgoing edge to a node N', labeled with $[[t_1/x_1, \ldots, t_m/x_m]s, q]$. By the construction, $\tau_{F,N}$ must be:

$$\bigwedge\{\tau_{t_1,N_1} \mid N_1 \in S_1\} \to \cdots \to \bigwedge\{\tau_{t_m,N_m} \mid N_m \in S_m\} \to q,$$

where S_i is the set of nodes where t_i occurs in a head position.

Let S'_i be the set of nodes N_i in S_i such that $\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} t_i : \tau_{t_i,N_i}$ occurs in the derivation of $\Pi_{[t_1/x_1,\ldots,t_m/x_m]s,N'}$. From the derivation tree $\Pi_{[t_1/x_1,\ldots,t_m/x_m]s,N'}$, we obtain a derivation for:

$$\Gamma_{\mathcal{C}} \cup \{x_i : \tau_{t_i,N_i} \mid N_i \in S'_i\} \vdash_{\mathcal{B}} s : q.$$

Since $S'_i \subseteq S_i$, we get

$$\Gamma_{\mathcal{C}} \cup \{x_i : \tau_{t_i,N_i} \mid N_i \in S_i\} \vdash_{\mathcal{B}} s : q$$

by weakening (Lemma 4.2). By using T-Abs, we obtain

$$\Gamma_{\mathcal{C}} \vdash_{\mathcal{B}} \lambda x_1, \ldots, x_m.s : \theta$$

as required. □

C. EXAMPLES USED IN EXPERIMENTS

This section explains other examples used in the experiments in Section 6.

`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)
```

It has been reduced to a recursion scheme, by combining the reduction discussed in Section 3.2 with the representation of exception handlers as additional continuation arguments (recall Section 3.3.3). The rules for `f` and `read` are as follows.

$$\begin{array}{l}
S \to \mathit{NewR}\ C_1 \\
C_1\ x \to \mathit{NewW}\ (C_2\ x) \\
C_2\ x\ y \to F\ x\ y\ (\mathit{Close}\ x\ (\mathit{Close}\ y\ \mathtt{end}))\ \mathtt{end} \\
F\ x\ y\ \mathit{ex}\ k \to \mathit{Read}\ x\ \mathit{ex}\ (\mathit{Write}\ y\ \mathit{ex}\ (F\ x\ y\ \mathit{ex}\ k)) \\
\mathit{Read}\ x\ \mathit{ex}\ k \to \mathit{ReadWithoutExn}\ x\ (\mathtt{br}\ \mathit{ex}\ k) \\
\mathit{ReadWithoutExn}\ x\ k \to x\ \mathit{read}\ k \\
\mathit{NewR}\ k \to \mathtt{br}\ (k\ K)\ (\nu^{q_{\mathit{read_only}}}\ (k\ I)) \\
\ldots
\end{array}$$

Here, the variable *ex* in the definition of F 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 recursion scheme `Lock1` has been obtained from 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 = _ in let x = newlock() in
(f b x; g b x)
```

Here, the resource usage specification is that every lock should be used according to $(\mathtt{lock} \cdot \mathtt{unlock})^*$. Booleans have been expressed by using Church encoding.

`Order5-2` is a recursion scheme obtained from 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 "foo" in
   gencon genr f
```

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