Untyped Recursion Schemes and Infinite Intersection Types

Takeshi Tsukada and Naoki Kobayashi

Tohoku University

Abstract. A new framework for higher-order program verification has been recently proposed, in which higher-order functional programs are modelled as higher-order recursion schemes and then model-checked. As recursion schemes are essentially terms of the *simply-typed* lambda-calculus with recursion and tree constructors, however, it was not clear how the new framework applies to programs written in languages with more advanced type systems. To circumvent the limitation, this paper introduces an *untyped* version of recursion schemes and develops an infinite intersection type system that is equivalent to the model checking of untyped recursion schemes, so that the model checking can be reduced to type checking as in recent work by Kobayashi and Ong for typed recursion schemes. The type system is undecidable but we can obtain decidable subsets of the type system by restricting the shapes of intersection types, yielding a sound (but incomplete in general) model checking algorithm.

1 Introduction

Model checking of recursion schemes [1, 2] has recently been applied to higher-order program verification [3, 4]. A recursion scheme is a grammar for generating a possibly infinite tree; from a programming language point of view, it is a term of the simply-typed λ-calculus with recursion and tree constructors. The idea of the higher-order program verification is to transform a higher-order program into a recursion scheme that generates a tree representing event sequences of the program, so that temporal properties of the program can be verified by model-checking the recursion scheme. The main advantage of the verification method is that it is sound and *complete* for a certain class of higher-order programs – the *simply-typed* λ-calculus with recursion and finite base types.

There is however a gap between the class of higher-order programs that can be handled directly by the above method (the *simply-typed* λ-calculus with finite base types) and the class of programs written in real programming languages. One of the main restrictions on the former class is that programs must be *simply-typed*. It was unclear how the method above can be applied to programs that use more advanced types, such as polymorphism and recursive types.

To address the problem above, we remove the restriction that recursion schemes must be simply-typed, by extending Kobayashi and Ong's type-based approach to model-checking recursion schemes [3, 5]. Instead of considering each extension of the simple type system (such as ML type system and System F with/without recursive types), we study the most expressive type system – an infinite intersection sort system for recursion schemes, and develop a type-based method for model-checking intersection-typed recursion schemes. (The infinite intersection sort system is the most expressive in the sense that all untyped recursion schemes that generate valid trees are typable in the type system.) Since various type systems, possibly with polymorphic and recursive types, can be regarded as restricted forms of the intersection type system, our studies of the intersection-typed recursion schemes can serve as a common foundation for recursion schemes extended with various type systems.

In the paper, in Section 3, we first give an intersection *sort* system that exactly characterizes the (untyped) recursion schemes that generate well-ranked trees, in the sense that a recursion scheme is well-sorted if and only if it generates a well-ranked tree. We then (in Section 4) introduce an intersection type system, parameterized by a Büchi automaton $\mathcal{A}$ with a trivial acceptance condition, such that a recursion scheme is well-typed if and only if it generates a well-ranked tree accepted by $\mathcal{A}$. One of the main results, presented in Section 5, is that if a recursion scheme is well-sorted under a sort environment Γ, the recursion scheme is well-typed if and only if it is so under a refinement of Γ. Thus, although the infinite intersection type system is undecidable in general, it is decidable whether, given a recursion scheme well-sorted under a *finite* sort environment, the recursion scheme is well-typed. As a consequence, the model checking of recursion schemes with ML-style polymorphism is decidable. Model checking of recursion schemes with more advanced type systems is undecidable, but we can still obtain a sound and decidable (but incomplete) type system by restricting the syntax of intersection types so that there can be finitely many refinements for each sort environment.

For the space restriction, we omit some proofs. They are found in a longer version available from `http://www.kb.ecei.tohoku.ac.jp/~tsukada/papers/fossacs10-full.pdf`.

2 Preliminaries

Trees Let $T \subseteq (\omega - \{0\})^*$ be a subset of finite sequences of natural numbers without 0. T is a *tree* if it satisfies the following conditions: (i) $\varepsilon \in T$, (ii) if $pi \in T$ for $p \in (\omega - \{0\})^*, i \in \omega - \{0\}$, then $p \in T$, and (iii) if $pi \in T$ for $p \in (\omega - \{0\})^*, i \in (\omega - \{0\})$, then $pj \in T$ for every $1 \leq j \leq i$. Note that 0 is not used as an index of trees; This convention makes some definitions and proofs simple. If $p, pi \in T$, we say p is the *parent* of pi and pi is a *child* of p. The *rank* of the node p, written $\#_T(p)$, is the cardinality of the set of its children. Note that for every $p \in T$, $\#_T(p) \leq \omega$. We omit the subscript T when it is clear from the context. We assume that the elements of the set $(\omega - \{0\})^*$ are ordered by the prefix ordering, i.e., $p_0 \leq p_1$ if and only if there is some p'_1 such that $p_1 = p_0 p'_1$.

Let A be an alphabet (i.e. a set of symbols). An *A-labeled tree* is a function $r : T \to A$ from a tree T to the alphabet A. Let Σ be a *ranked alphabet*, i.e.

a map from an alphabet A to $\omega \cup \{\omega\}$. Σ is *finitely ranked* if $\Sigma(a) < \omega$ for every $a \in dom(\Sigma)$. By abuse of notation, we often write $a \in \Sigma$ for $a \in dom(\Sigma)$. A $dom(\Sigma)$-labeled tree r is also called a Σ-labeled tree. It is *well-ranked* if for every $p \in dom(r)$, $\Sigma(r(p)) = \#p$.

Trivial Automata A Büchi tree automaton $\mathcal{A}$ with a trivial acceptance condition (called a *trivial automaton*, for short) is a quadruple (Q, Σ, q_S, Δ)[1] where Q is a finite set of states, Σ is a finitely-ranked alphabet, $q_S \in Q$ is an initial state, and $\Delta \subseteq Q \times \Sigma \times Q^*$ is a transition relation. The transition relation must respect the rank, i.e., if $(q, a, q_1, \ldots, q_n) \in \Delta$, then $n = \Sigma(a)$. A trivial automaton is *deterministic* if, for each pair $(q, a) \in Q \times dom(\Sigma)$, there is at most one element of the form $(q, a, q_1, \ldots, q_n)$ in Δ.

For a well-ranked Σ-labeled tree $r : T \to \Sigma$, a *run* of $\mathcal{A}$ on r is a Q-labeled tree $\varrho : T \to Q$, satisfying $(\varrho(p), r(p), \varrho(p1), \ldots, \varrho(pn)) \in \Delta$ for each $p \in T$, where $n = \Sigma(r(p))$.

For a state q, a Σ-labeled well-ranked tree r is *accepted by $\mathcal{A}$ from the state* q, if there is a run ϱ of $\mathcal{A}$ that satisfies $\varrho(\varepsilon) = q$. We write $\mathcal{L}_{\mathcal{A}}(q)$ for the set of trees accepted from q. The *language recognized by $\mathcal{A}$*, written $\mathcal{L}_{\mathcal{A}}$, is $\mathcal{L}_{\mathcal{A}}(q_S)$.

We assume that there is one distinguished element $\perp \in \Sigma$ with $\Sigma(\perp) = 0$, and for any state q of any trivial automata, $(q, \perp) \in \Delta$. Intuitively, $\perp$ is the undefined tree, which is accepted from any state.

Recursion Schemes An *(untyped) recursion scheme* $\mathcal{G}$ is a quadruple $(\Sigma, \mathcal{N}, \mathcal{R}, S)$, where: (i) Σ is a ranked alphabet with a distinguished element $\perp$ of rank 0. An element of Σ is called a *terminal*; (ii) $\mathcal{N}$ is a finite set of symbols called *non-terminals*; (iii) $S \in \mathcal{N}$ is the *start symbol*; and (iv) $\mathcal{R}$ is a set of rewriting rules of the form $\{F_1 \tilde{x_1} \to t_1, \ldots, F_n \tilde{x_n} \to t_n\}$. Here, F_i is a nonterminal, $\tilde{x}$ abbreviates a sequence of variables and t_i is an applicative term over $\mathcal{N} \cup (\Sigma - \{\perp\}) \cup \{\tilde{x_i}\}$ (i.e. a term constructed from $\mathcal{N} \cup (\Sigma - \{\perp\}) \cup \{\tilde{x_i}\}$ and applications). There must be exactly one rule for each non-terminal. If $(F\tilde{x} \to t) \in \mathcal{R}$, we write $\mathcal{R}(F) = \lambda\tilde{x}.t$, where $\lambda\tilde{x}$ abbreviates a sequence of lambda abstractions. We identify an applicative term over X with an X-labeled tree in the standard manner: a term $x\, t_1 \ldots t_n$ (where $x \in X$) as a tree whose root is labeled by x, having $t_1, \cdots, t_n$ as subtrees.

In the definition of standard recursion schemes [2], there are additional conditions that each symbol or variable is assigned a simple type (with o, the type of trees, as a unique base type), and that both the left- and right-hand sides of each rule must have type o. In this paper, we call the standard recursion schemes *(simply-)typed recursion schemes*, and call recursion schemes without the typing constraint *untyped recursion schemes* or simply *recursion schemes*.

The rewriting relation $\longrightarrow_{\mathcal{G}}$ is defined inductively by: (i) $F\tilde{s} \longrightarrow_{\mathcal{G}} [\tilde{s}/\tilde{x}]t$, if $\mathcal{R}(F) = \lambda\tilde{x}.t$, and (ii) if $t \longrightarrow_{\mathcal{G}} t'$, then $ts \longrightarrow_{\mathcal{G}} t's$ and $st \longrightarrow_{\mathcal{G}} st'$.

For a term t, we define $t^{\perp}$ by: (i) $a^{\perp} = a$ for each terminal a; (ii) $(t_1 t_2)^{\perp} = t_1^{\perp}\ t_2^{\perp}$ if $t_1^{\perp} \neq \perp$; and (iii) $t^{\perp} = \perp$ otherwise. The partial order $\sqsubseteq$ on Σ is defined

[1] A trivial automaton is a Büchi automaton where all the states are final.

as $a \sqsubseteq b$ if and only if $a = b$ or $a = \bot$. We extend this ordering to the ordering on Σ-labeled trees by: $r_1 \sqsubseteq r_2$ if and only if $T_1 \subseteq T_2$ and $r_1(p) \sqsubseteq r_2(p)$ for every $p \in T_1$. The *value tree* of $\mathcal{G}$, written $[\![\mathcal{G}]\!]$, is $\bigsqcup\{t^{\bot} \mid S \longrightarrow^*_{\mathcal{G}} t\}$, where $\bigsqcup S$ is the least upper bound with respect to $\sqsubseteq$. The value tree is always well-defined, as the rewriting relation is confluent.

In this paper, we are interested in the model checking problem:
"Given a recursion scheme $\mathcal{G}$ and a trivial automaton $\mathcal{A}$, does $[\![\mathcal{G}]\!] \in \mathcal{L}_{\mathcal{A}}$ hold?" In the case of *simply-typed* recursion schemes, the problem is known to be decidable [2].[2] In the case of *untyped* recursion schemes, however, the model checking problem above (or, even the problem of checking whether the value tree is well-ranked) is undecidable, as the untyped recursion schemes are essentially terms of the untyped λ-calculus (with uninterpreted function symbols).

Remark 1. The reader may wonder why we have recursion as primitives, even though a fixed-point combinator can be encoded in the untyped λ-calculus. That is because we later impose various type constraints corresponding to those of advanced type systems (e.g. a type system with ML polymorphism), in which a fixed-point combinator may no longer be encoded. Note that our main interests are in extending Kobayashi and Ong's type-based method [3, 5] for model-checking simply-typed recursion schemes to handle recursion schemes *with various advanced type systems*, and that we study infinite intersection type systems for untyped recursion schemes to establish common foundations.

3 Infinite Intersection Sorts

The goal of this section is to characterize the class of recursion schemes whose value trees are well-ranked. For this purpose, we construct an intersection type system in which a recursion scheme is well-typed if, and only if, the value trees of recursion schemes are well-ranked. In the following, we call this type system the *sort system* to avoid confusion with the type system for model checking introduced in Section 4.

In the sort system, intersection types are infinite both in width (i.e. we allow $\bigwedge_{i<\omega} \kappa_i$) and in depth (i.e. we allow sorts having infinite paths, like $o \to o \to o \to \cdots$). Note that such infinite intersection types are necessary for the complete characterization of recursion schemes that generate well-ranked trees: See Remark 2.

As defined below, a *sort* is a (possibly infinite) tree labeled by o, $\to$, and $\wedge$. The sort o (i.e. a tree consisting of a single node labelled by o) describes trees. The other constructors $\to$ and $\wedge$ describe functions and intersections as usual; for example, $(\mathsf{o} \wedge (\mathsf{o} \to \mathsf{o})) \to \mathsf{o}$ describes a function that takes as input a term that can be both used as a tree and a tree function, and returns a tree.

Definition 1 (sorts). *Let $K = \{(\mathsf{o}, 0), (\to, 2)\} \cup \{(\wedge^{\alpha}, \alpha) \mid \alpha \leq \omega\}$ be a ranked alphabet. A sort κ is a well-ranked K-labeled tree that satisfies: (i) $\kappa(\varepsilon) \in \{\mathsf{o}, \to$*

[2] Ong [2] proved the decidability for a more general case, where $\mathcal{A}$ is an alternating parity tree automaton.

}; (ii) If $\kappa(p) = \rightarrow$, then $\kappa(p1) = \wedge^\alpha$ and $\kappa(p2) \in \{\mathtt{o}, \rightarrow\}$; and (iii) If $\kappa(p) = \wedge^\alpha$, then $\kappa(pi) \in \{\mathtt{o}, \rightarrow\}$ for every i such that $pi \in dom(\kappa)$.

We often omit the superscript and simply write $\wedge$ for $\wedge^\alpha$. When κ_i's ($i < \alpha$) are sorts, we write $\bigwedge_{i<\alpha} \kappa_i$ for the tree whose root is labelled by $\wedge^\alpha$ and whose children are κ_i's. We write $\top$ for the empty intersection $\bigwedge_{i<0} \kappa_i$. Similarly, we write $\bigwedge_{i<\alpha} \kappa_i \rightarrow \kappa$ for the sort whose root is labelled by $\rightarrow$, and whose children are $\bigwedge_{i<\alpha} \kappa_i$ and κ. When $\alpha = 1$, we just write $\kappa_0 \rightarrow \kappa$ for $\bigwedge_{i<\alpha} \kappa_i \rightarrow \kappa$ (e.g. $(\mathtt{o} \rightarrow \mathtt{o}) \rightarrow \mathtt{o}$ for $\bigwedge_{i<1} \kappa_i \rightarrow \mathtt{o}$ where $\kappa_0 = \mathtt{o} \rightarrow \mathtt{o}$). We give a higher precedence to $\wedge$ than to $\rightarrow$.

The three conditions in the definition above are imposed just for a technical convenience (more specifically, for removing the introduction and elimination rules for intersection types). Note that, for example, $(\kappa_1 \rightarrow \kappa_2) \rightarrow \kappa_3$ (which is prohibited by the restriction (ii)) can be represented as $\bigwedge_{i<1} \kappa'_i \rightarrow \kappa_3$, where $\kappa'_0 = \kappa_1 \rightarrow \kappa_2$. A similar restriction on the syntax of types has been used for finite intersection type systems [6].

A type environment, denoted by Γ, is a (possibly infinite) set of bindings of the form $x{:}\tau$ (where non-terminals of recursion schemes are treated as variables). We often omit the curly brackets, and simply write $x_1 : \kappa_1, \ldots, x_n : \kappa_n$ for $\{x_1 : \kappa_1, \ldots, x_n : \kappa_n\}$. Note that we allow multiple bindings for the same variable, as in $\{x : \kappa_1, x : \kappa_2\}$. We abbreviate $\{x : \kappa_i \mid i < \alpha\}$ as $x : \bigwedge_{i<\alpha} \kappa_i$. We write $dom(\Gamma)$ for the set $\{x \mid \exists \tau.(x : \tau \in \Gamma)\}$.

We fix a finitely ranked alphabet Σ below. The typing rules for λ-terms are given as follows:

$$\frac{}{\Gamma, x : \kappa \vdash x : \kappa} \qquad \frac{\Gamma \vdash t_1 : \bigwedge_{i<\alpha} \kappa_i \rightarrow \kappa \quad \Gamma \vdash t_2 : \kappa_i (\text{for every } i < \alpha)}{\Gamma \vdash t_1 t_2 : \kappa}$$

$$\frac{}{\Gamma \vdash a : \underbrace{\mathtt{o} \rightarrow \cdots \rightarrow \mathtt{o}}_{\Sigma(a)} \rightarrow \mathtt{o}}$$

$$\frac{}{\Gamma \vdash \lambda x.t : o} \qquad \frac{\Gamma \cup \{x : \kappa_i \mid i < \alpha\} \vdash t : \kappa \quad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.t : \bigwedge_{i<\alpha} \kappa_i \rightarrow \kappa}$$

The rules above are standard, except the rule on the left bottom. It is due to our definition of the value tree of a recursion scheme. To see why, consider the recursion scheme $\mathcal{G}$, consisting of the two rules $S \rightarrow F$ and $F\,x \rightarrow \mathtt{a}$. The rewriting of S gets stuck as $S \rightarrow F$, so that the value tree of $\mathcal{G}$ is $\perp$. Since $\perp$ is a well-ranked tree, F (which is essentially $\lambda x.\mathtt{a}$) should be assigned type $\mathtt{o}$.

A recursion scheme $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{R}, S)$ is well-typed under Γ, written $\vdash \mathcal{G} : \Gamma$, if $dom(\Gamma) \subseteq \mathcal{N}$, $\Gamma \vdash \mathcal{R}(F) : \tau$ holds for every $F : \tau \in \Gamma$, and $S : \mathtt{o} \in \Gamma$. We write $\vdash \mathcal{G}$ if there exists Γ such that $\vdash \mathcal{G} : \Gamma$.

The following theorem states soundness and completeness of the sort system.

Theorem 1. *For any recursion scheme $\mathcal{G}$, $[\![\mathcal{G}]\!]$ is well-ranked if and only if $\vdash \mathcal{G}$.*

Proof. A special case of Theorems 2 and Theorem 3 in Section 4, where the automaton $\mathcal{A}$ is $(\{o\}, \Sigma, o, \Delta)$ such that $(o, a, \overbrace{o, \ldots, o}^{n}) \in \Delta$ for every $a \in \Sigma$ of rank n. □

Remark 2. As already mentioned, intersection sorts in our type system may be infinite both in width and depth, and non-regular (i.e. may not be expressed by finite recursive types). The following observations explain why simpler intersection sorts are insufficient. First, intersection types that are infinite in width but *finite in depth* guarantee that λ-terms are strongly normalizing [7]; thus the Y combinator (which can be defined by $Y\,f \to (A\,f)(A\,f)$ and $A\,f\,x \to f\,(x\,x)$) would not be typable. Secondly, the restriction of intersection sorts to finite recursive types (i.e. sorts that are finite in width and infinite but regular in depth) is also insufficient. This is because under this restriction, the typability would be recursively enumerable,[3] but the well-rankedness of the tree generated by an untyped recursion scheme is not. The latter can be easily proved, for example, by a reduction from the halting problem of a Minsky machine [8]. A Minsky machine consists of two counters holding natural numbers, and has instructions for counter increment/decrement, conditional/unconditional jumps, and halting [8]. We can use the standard Church encoding for natural numbers and booleans to simulate those instructions, and replace the halt command with a term generating an ill-ranked tree. The resulting untyped recursion scheme generates a well-ranked tree if and only if the Minsky machine does not halt.

4 Type System for Model Checking Untyped Recursion Schemes

In this section, we extend Kobayashi's type system [3] (for model-checking recursion schemes wrt safety properties) to deal with untyped recursion schemes.

Let $\mathcal{A} = (Q, \Sigma, q_S, \Delta)$ be a trivial automaton. We shall extend the sort system of the previous section by refining the sort o into a type of the form $\bigwedge\{q_1, \ldots, q_k\}$ where $q_1, \ldots, q_k \in Q$. Intuitively, a state q of the automaton is regarded as a type that describes the trees accepted by $\mathcal{A}$ from the state q, i.e., t has type q if and only if $[\![t]\!] \in \mathcal{L}_{\mathcal{A}}(q)$.

Definition 2 (Types). *Let $T = \{(q, 0) \mid q \in Q\} \cup \{(\to, 2)\} \cup \{(\wedge^{\alpha}, \alpha) \mid \alpha \leq \omega\}$ be the set of ranked alphabets. A type τ is a well-ranked T-labeled tree that satisfies the following conditions: (i) $\tau(\varepsilon) \neq \wedge$ (ii) If $\tau(p) = \to$, then $\tau(p1) = \wedge$, $\tau(p2) \neq \wedge$, and (iii) If $\tau(p) = \wedge$, then $\tau(pi) \neq \wedge$ for every $i \in \omega - \{0\}$.*

The typing rules are almost the same as the sorting rules, except that the type of a terminal is determined by the transition function of the automaton.

$$\frac{}{\Gamma, x : \tau \vdash_{\mathcal{A}} x : \tau} \qquad \frac{(q, a, q_1, \ldots, q_n) \in \Delta}{\Gamma \vdash_{\mathcal{A}} a : q_1 \to \cdots \to q_n \to q} \qquad \frac{}{\Gamma \vdash_{\mathcal{A}} \lambda x.t : q}$$

[3] Note that since the set of finite recursive sorts is recursively enumerable, the set of valid type judgements is also recursively enumerable.

$$\frac{\Gamma \vdash_{\mathcal{A}} t_1 : \bigwedge_{i<\alpha} \tau_i \to \tau \qquad \Gamma \vdash_{\mathcal{A}} t_2 : \tau_i \text{ for all } i < \alpha}{\Gamma \vdash_{\mathcal{A}} t_1\, t_2 : \tau} \qquad \frac{\Gamma \cup \{x : \tau_i \mid i < \alpha\} \vdash_{\mathcal{A}} t : \tau \qquad x \notin dom(\Gamma)}{\Gamma \vdash_{\mathcal{A}} \lambda x.t : \bigwedge_{i<\alpha} \tau_i \to \tau}$$

We write $\vdash_{\mathcal{A}} \mathcal{G} : \Gamma$ if (i) $dom(\Gamma) \subseteq \mathcal{N}$, (ii) $\Gamma \vdash_{\mathcal{A}} \mathcal{R}(F) : \tau$ holds for every $F : \tau \in \Gamma$, and (iii) $S : q_S \in \Gamma$.

4.1 Soundness of the Type System

We use the following lemma to establish the soundness of our type system.

Lemma 1 (Subject Reduction). *Suppose $\vdash_{\mathcal{A}} \mathcal{G} : \Gamma$. If $\Gamma \vdash_{\mathcal{A}} t : \tau$ and $t \to_{\mathcal{G}}^* t'$, then $\Gamma \vdash_{\mathcal{A}} t' : \tau$.*

Theorem 2 (Soundness). *Let $\mathcal{G}$ be a recursion scheme and $\mathcal{A}$ be a trivial automaton. If $\vdash_{\mathcal{A}} \mathcal{G}$, then $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{A}$.*

Proof Sketch It suffice to show that if $S \to_{\mathcal{G}}^* t$, then $t^{\perp}$ is accepted by $\mathcal{A}$. Assume $S \to_{\mathcal{G}}^* t$. By the definition of $\vdash_{\mathcal{A}} \mathcal{G}$, there is a type environment such that $\vdash_{\mathcal{A}} \mathcal{G} : \Gamma$. By Lemma 1, we get $\Gamma \vdash_{\mathcal{A}} t : q_S$, from which $t^{\perp} \in \mathcal{L}_{\mathcal{A}}$ follows. $\square$

4.2 Completeness of the Type System

We show that our type system is complete in the sense that a recursion scheme is well-typed if its value tree is accepted by $\mathcal{A}$. The overall idea of the proof is similar to the completeness proof of Kobayashi and Ong's type system for the modal μ-calculus model checking of typed recursion schemes [5]; Type information is extracted from a reduction sequence of the recursion scheme, by observing how each non-terminal is used in the reduction. Some non-trivial adjustments are necessary, however, since sorts (which are called kinds in [5]) are infinite (thus, we cannot use induction on sorts unlike in [5]).

Theorem 3 (Completeness). *Let $\mathcal{G}$ be a recursion scheme and $\mathcal{A}$ be a trivial automaton. If $[\![\mathcal{G}]\!]$ is well-ranked and accepted by $\mathcal{A}$, then $\vdash_{\mathcal{A}} \mathcal{G}$.*

We fix below a recursion scheme $\mathcal{G}$ and a trivial automaton $\mathcal{A}$. By the assumption that $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{A}$, there is a run tree of $\mathcal{A}$ over $[\![\mathcal{G}]\!]$ whose root is labeled by q_S. We fix such a run tree r below.

We shall define a *reduction tree* $\mathcal{T}^{\infty}$, which expresses the process of the value tree of $\mathcal{G}$ being constructed in a reduction sequence. In the reduction tree, each term is annotated with a label to keep track of the origin of the term. The set Term^L of *annotated terms*, ranged over by u, is given by:

$$u ::= x^l \mid a^l \mid F^l \mid (u_1 u_2)^l,$$

where $l \subseteq \omega^* \times \omega$. Intuitively, u^l with $(p, n) \in l$ means that the term u has occurred in the node p of $\mathcal{T}^{\infty}$, in the form $c \cdots u\, u_{n-1} \cdots u_1$ (where c is a non-terminal or a terminal). We often write u^l for u when the outermost label is l.

We define $(u^l)^{+(p,i)}$ as $u^{l\cup(p,i)}$. We write $\flat(u)$ for the term obtained by removing all the labels from u. We omit some of the labels of an annotated term when they are not important. The substitution of annotated terms is defined as a homomorphism satisfying $[v^l/x](x^{l'}) = v^{l\cup l'}$.

The *expansion relation* $\rhd$ on (finite and unranked) $(\omega^* \times \mathrm{Term}^L)$-labelled trees is the least relation that satisfies the following condition.

If T is a $(\omega^* \times \mathrm{Term}^L)$-labelled tree with $T(p) = (p', c\, u_n \dots u_1)$, where $c \in \Sigma \cup \mathcal{N}$, and there is no child of p in T, then:

1. If $c = F^l$ and $\mathcal{R}(F) = \lambda x_n \dots x_k.s$ and $k \geq 1$, then
$$T \rhd T \cup \{(p1, (p', ([u_n^{+(p,n)} \dots u_k^{+(p,k)}/x_n \dots x_k]s)u_{k-1}^{+(p,k-1)} \dots u_1^{+(p,1)}))\}$$
2. If $c = a^l$ (which implies $n = \Sigma(a)$), then:
$$T \rhd T \cup \{(pi, (p'i, u_{n-i+1}^{+(p,n-i+1)}))) \mid 1 \leq i \leq n\}$$

In a label of the form (p', u), the annotated term u represents the term being reduced, and p' represents the corresponding position in the value tree of $\mathcal{G}$. In other words, the subtree of $[\![\mathcal{G}]\!]$ at position p' will be generated by reducing u.

The *reduction tree* $\mathcal{T}^\infty$ is the $(\omega^* \times \mathrm{Term}^L)$-labelled tree obtained by an infinite expansion of $(\epsilon, S^{\{\}})$, i.e. $\mathcal{T}^\infty = \bigcup\{T \mid \{\ (\epsilon, S^{\{\}})\ \} \rhd^* T\}$.

It is easy to see that $\mathcal{T}^\infty$ is well-defined. Note that $V = \{(p', a) \mid \exists p \in \omega^*, \mathcal{T}^\infty(p) = (p', a\, u_n \dots u_1))\}$ is essentially equal to the value tree $[\![\mathcal{G}]\!]$. The only difference is that if $[\![\mathcal{G}\ ]\!]\ (p') = \bot$, then $V(p')$ is undefined.

Example 1. Let $\mathcal{G}_0 = (\Sigma, \{S, F\}, \mathcal{R}, S)$, where $\Sigma = \{(\mathtt{br}, 2), (\mathtt{a}, 1), (\mathtt{b}, 1), (\mathtt{c}, 0)\}$, $\mathcal{R} = \{S \to F\ \mathtt{c},\ F\ x \to \mathtt{br}\ \mathtt{c}\ (\mathtt{a}(F(\mathtt{b}(x))))\}$. The reduction tree $\mathcal{T}^\infty$ is shown in Figure 4.2. We have $\mathcal{T}^\infty(1121) = (21, \cdots \mathtt{c}^{(1,1)} \cdots)$, which means this occurrence $\mathtt{c}$ appears at 1 as the 1st argument (counting from right to left). The node at 112 is of the form $(2, \mathtt{a}\, u)$, which means $[\![\mathcal{G}_0\ ]\!]\ (2) = \mathtt{a}$. □

Lemma 2. *Let $u_1^{l_1}$ be an annotated term occurring in $\mathcal{T}^\infty$, i.e., there exists a path p_1 such that $\mathcal{T}^\infty(p_1) = (p_1', u)$ and $u_1^{l_1}$ is a subterm of u. If $(p, n) \in l_1$ then $\mathcal{T}^\infty(p) = (p', v\, v_n^l \dots v_1)$ with $u_1^{l_1} = v_n^{l\cup l'}$ for some l'.*

We define $\theta_{(\wedge,p,i)}$ and $\theta_{(\to,p,i)}$ as the (possibly infinite) trees that satisfy the following equations.

$$\theta_{(\wedge,p,i)} = \bigwedge\{\theta_{(\to,p_0,n)} \mid \mathcal{T}^\infty(p_0) = (p_0', u^l\, u_n \dots u_1) \text{ and } (p, i) \in l\}$$
$$\theta_{(\to,p,i)} = \theta_{(\wedge,p,i)} \to \cdots \to \theta_{(\wedge,p,1)} \to r(p')$$
$$\text{where } \mathcal{T}^\infty(p) = (p', u) \text{ and } r \text{ is the run tree.}$$

In the first equation, we assume that $\theta_{(\to,p_0,i)}$'s are ordered according to a certain linear order among elements of $(\omega - \{0\})^*$. Thus, $\theta_{(\wedge,p,i)}$ and $\theta_{(\to,p,i)}$ are uniquely determined, and every $\theta_{(\to,p,i)}$ is a type.

Let $\Gamma_\mathcal{G}^r$ be $\{F : \theta_{(\to,p,n)} \mid \mathcal{T}^\infty(p) = (p', F^l\, u_n \dots u_1)\}$. We show that $\Gamma_\mathcal{G}^r$ is the witness of well-typing of $\mathcal{G}$, i.e. $\vdash \mathcal{G} : \Gamma_\mathcal{G}^r$.

We first show that each term occurring in $\mathcal{T}^\infty$ is well-typed under $\Gamma_\mathcal{G}^r$.

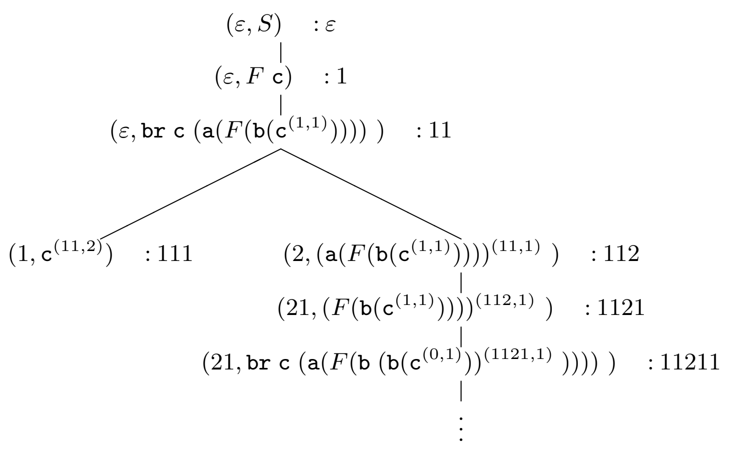
(ε, S) : ε
(ε, F c) : 1
(ε, br c (a(F(b(c^(1,1)))))) : 11
(1, c^(11,2)) : 111
(2, (a(F(b(c^(1,1)))))^(11,1)) : 112
(21, (F(b(c^(1,1)))))^(112,1)) : 1121
(21, br c (a(F(b (b(c^(0,1)))^(1121,1))))) : 11211

Fig. 1. $\mathcal{T}^\infty$ for $\mathcal{G}_0$. We omit the empty annotation. . Here p of $(p_0, u) : p$ is the path of the node (p_0, u), hence they are not a part of the nodes.

Lemma 3. *If $\mathcal{T}^\infty(p) = (p', uu_n \dots u_1)$, then $\Gamma^r_\mathcal{G} \vdash \flat(u) : \theta_{(\rightarrow,p,n)}$.*

The following lemma is a kind of the inverse of the substitution lemma; we derive a typing for t from that of $[s_1^{l_1} \dots s_k^{l_k}/x_1 \dots x_{l_k}]t$. Recall that $\Gamma, x : \bigwedge_{i<\alpha} \tau_i \vdash_\mathcal{A} t : \tau$ is the abbreviation of $\Gamma, x : \tau_0, x : \tau_1, \dots \vdash_\mathcal{A} t : \tau$. We define $\Gamma \vdash_\mathcal{A} t : \bigwedge_{i<\alpha} \tau_i$ as the abbreviation of $\Gamma \vdash_\mathcal{A} t : \tau_i$ for all $i < \alpha$.

Lemma 4. *Suppose that $\mathcal{T}^\infty(p) = (p', u\, u_n \dots u_1)$ and $u = [s_1^{l_1} \dots s_k^{l_k}/x_1 \dots x_k]v$, with $(p_i, j_i) \in l_i$ for each $i \in \{1, \dots, k\}$. Let Γ be $x_1 : \theta_{(\wedge,p_1,j_1)}, \dots, x_k : \theta_{(\wedge,p_k,j_k)}$. Then $\Gamma^r_\mathcal{G}, \Gamma \vdash_\mathcal{A} \flat(v) : \theta_{(\rightarrow,p,n)}$.*

Proof. By induction on the structure of v.

– Case where v does not contain x_i for any i: Immediate from Lemma 3.

– Case $v = x_i$: Note that $\mathcal{T}^\infty(p) = (p', s_i^{l_i}\, u_1 \dots u_n)$. By the assumption $(p_i, j_i) \in l_i$ and the definition of $\theta_{(\wedge,p_i,j_i)}$, we have $x : \theta_{(\rightarrow,p,n)} \in \Gamma$. So, by using (T-Var), we obtain $\Gamma^r_\mathcal{G}, \Gamma \vdash x_i : \theta_{(\rightarrow,p,n)}$ as required.

– Case $v = v_{00}v_{01}$: Let $u_{0j} = [s_1 \dots s_k/x_1 \dots x_k]v_j$ for $j = 0, 1$ and l be the outermost label of v_{01}, i.e., v_{01}^l. Note that $u = u_{00}u_{01}$ and $\mathcal{T}^\infty(p) = (p', u_{00}^{l_{00}} u_{01}^{l_{01}} u_n \dots u_1)$. By the induction hypothesis, $\Gamma^r_\mathcal{G}, \Gamma \vdash_\mathcal{A} \flat(v_{00}) : \theta_{(\rightarrow,p,n+1)}$. Because $\theta_{(\rightarrow,p,n+1)} = \theta_{(\wedge,p,n+1)} \rightarrow \theta_{(\rightarrow,p,n)}$, what we should show is $\Gamma^r_\mathcal{G}, \Gamma \vdash_\mathcal{A} \flat(v_{01}) : \theta_{(\wedge,p,n+1)}$. Let p_0 be any path such that $\mathcal{T}^\infty(p_0) = (p'_0, v'^{l'}\, v'_m, \dots, v'_1)$ and $(p, n+1) \in l'$. By Lemma 2, we obtain $v'^{l'} = u_{01}^{l_{01} \cup l'_{01}}$. Therefore $v'^{l'} = [s_1 \dots s_k/x_1 \dots x_k](v_{01}^{l \cup l'_{01}})$. By using induction hypothesis, we have $\Gamma^r_\mathcal{G}, \Gamma \vdash_\mathcal{A} \flat(v_{01}) : \theta_{(\rightarrow,p_0,m)}$. So by the definition of $\theta_{(\wedge,p,n+1)}$, we obtain $\Gamma^r_\mathcal{G}, \Gamma \vdash_\mathcal{A} \flat(v_{00}) : \theta_{(\wedge,p,n+1)}$ as required. □

We are now ready to prove the completeness of the type system.

Proof of Theorem 3. It is easy to see that $S : q_S \in \Gamma^r_{\mathcal{G}}$. Thus, it remains to show that $\Gamma^r_{\mathcal{G}} \vdash \mathcal{R}(F) : \tau$ holds for each $F : \tau \in \Gamma^r_{\mathcal{G}}$. By the construction of $\Gamma^r_{\mathcal{G}}$, there is a path p that satisfies $\mathcal{T}^\infty(p) = (p', F^l u_n \dots u_1)$ and $\tau = \theta_{(\wedge,p,n)} \to \cdots \to \theta_{(\wedge,p,1)} \to r(p')$. Suppose $F : \tau \in \Gamma^r_{\mathcal{G}}$ and $\mathcal{R}(F) = \lambda x_n \lambda x_{n-1} \dots \lambda x_k.s$ (here, k may be a negative integer).

– Case $k \leq 0$: By using (T-Bot), we obtain $\Gamma^r_{\mathcal{G}} \cup \{x_i : \theta_{(\wedge,p,i)} \mid 1 \leq i \leq n\} \vdash \lambda x_0 \lambda x_{-1} \dots \lambda x_k.s : r(p')$. By using (T-Abs), we obtain $\Gamma^r_{\mathcal{G}} \vdash \lambda x_n \dots \lambda x_k : \tau$.

– Case $k > 0$: By the definition of $\mathcal{T}^\infty$, we have:

$$\mathcal{T}^\infty(p1) = (p', ([u_n^{+(p,n)} \dots u_k^{+(p,k)}/x_n \dots x_k]s) u_{k-1}^{(p,k-1)} \dots u_1^{(p,1)}).$$

By Lemma 4, $\Gamma^r_{\mathcal{G}}, \{x_i : \theta_{(\wedge,p,i)} \mid k \leq i \leq n\} \vdash s : \theta_{(\wedge,p,k-1)} \to \cdots \to \theta_{(\wedge,p,1)} \to r(p')$. By using (T-Abs), we obtain $\Gamma^r_{\mathcal{G}} \vdash \lambda x_n \dots \lambda x_k.s : \theta_{(\wedge,p,n)} \to \cdots \to \theta_{(\wedge,p,1)} \to r(p')$ as required. $\square$

5 Theory of Refinement

The purpose of this section is to develop a theory for applying our type system to verification of recursion schemes that model programs written in *typed* programming languages. The soundness and completeness theorems in the previous section imply that our type system for untyped recursion schemes is undecidable in general (as the model-checking problem is undecidable). Note, however, that a recursion scheme obtained from a typed program is well-typed under a certain type system. Thus, we are interested in the model checking problem: "Given a recursion scheme $\mathcal{G}$ *that is well-typed in a type system* T, is the value tree $\mathcal{G}$ accepted by $\mathcal{A}$?" Note that the type system T may ensure, in addition to the well-rankedness, certain properties of the value tree, like "every child of the node labelled by *cons* is *true* or *false*" (e.g. when the source program has type **bool list**). Thus, the type system T can be regarded as a restricted form of the intersection type system $\mathcal{T}_{\mathcal{A}'}$ for another automaton $\mathcal{A}'$. Therefore, the above model checking problem is refined to:

> Given a recursion scheme $\mathcal{G}$ *well-typed in a certain restriction of* $\mathcal{T}_{\mathcal{A}'}$, is the value tree $\mathcal{G}$ accepted by $\mathcal{A}$ (or equivalently, is $\mathcal{G}$ well-typed in $\mathcal{T}_{\mathcal{A}}$)?

Because of the assumption that $\mathcal{G}$ is well-typed, the model checking problem can be solved in certain cases. For example, Kobayashi and Ong's work [3, 5] can be considered as studies of a special case of the problem above, where $\mathcal{A}'$ is the Büchi automaton with a single state $\mathtt{o}$, and sorts are restricted to simple and finite ones (without intersections). The main result of this section (Theorem 5) is that if $\mathcal{G}$ is well-typed in $\mathcal{T}_{\mathcal{A}'}$, then $\mathcal{G}$ is well-typed in $\mathcal{T}_{\mathcal{A}}$ if, and only if, $\mathcal{G}$ is so under certain restricted type environments of $\mathcal{T}_{\mathcal{A}}$, so that the model checking problem can sometimes be solved effectively. Below we focus on the case where there is an *automata homomorphism* from $\mathcal{A}$ to $\mathcal{A}'$.

Definition 3. *Let* $\mathcal{A}_1 = (Q^1, \Sigma, q^1_S, \Delta^1)$ *and* $\mathcal{A}_2 = (Q^2, \Sigma, q^2_S, \Delta^2)$ *be trivial automata. A homomorphism* $f : \mathcal{A}_1 \to \mathcal{A}_2$ *of automata is a map from the states* *of $\mathcal{A}_1$ to the states of $\mathcal{A}_2$ satisfying the following conditions: (i) f maps the initial state to the initial state, i.e. $f(q_S^1) = q_S^2$. (ii) f respects the transitions, i.e. $(q, a, q_1, \ldots, q_n) \in \Delta^1$ implies $(f(q), a, f(q_1), \ldots, f(q_n)) \in \Delta^2$.*

For a given homomorphism $f : \mathcal{A}_1 \to \mathcal{A}_2$ of automata, we extend this function to a map $\hat{f}$ from types of $\mathcal{T}_{\mathcal{A}_1}$ to those of $\mathcal{T}_{\mathcal{A}_2}$, by:

$$\hat{f}(\tau)(p) = \begin{cases} f(\tau(p)) & (\text{if } \tau(p) \in Q^1) \\ \tau(p) & (\text{if } \tau(p) = \to \text{ or } \wedge) \end{cases}$$

and we define $\hat{f}(\Gamma)$ by $\hat{f}(\{F_1 : \tau_1, \ldots\}) = \{F_1 : \hat{f}(\tau_1), \ldots\}$. Then we can see that any homomorphism f preserves typability.

Theorem 4. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be automata, $\mathcal{G}$ be a recursion scheme and $f : \mathcal{A}_1 \to \mathcal{A}_2$ be a homomorphism of automata. If $\Gamma \vdash_{\mathcal{A}_1} \mathcal{G}$, then $f(\Gamma) \vdash_{\mathcal{A}_2} \mathcal{G}$.*

Note that the sort system given in Section 3 is the same as $\mathcal{T}_{\mathcal{A}^\top}$, where $\mathcal{A}^\top$ is the trivial automaton $(\{o\}, \Sigma, o, \Delta)$ such that $(o, a, \overbrace{o, \ldots, o}^{n}) \in \Delta$ for every $a \in \Sigma$ of rank n. Moreover, for any trivial automaton $\mathcal{A}$, $f : \mathcal{A} \to \mathcal{A}^\top : q \mapsto o$ is a (unique) homomorphism.

The main goal of this section is to establish a "backward" property. Given a homomorphism $f : \mathcal{A}_1 \to \mathcal{A}_2$, we want to derive a property about the typability of $\mathcal{G}$ in $\mathcal{T}_{\mathcal{A}_1}$, assuming that $\Gamma \vdash_{\mathcal{A}_2} \mathcal{G}$. For that purpose, we define a *refinement relation* $\sqsubseteq_f$ between types of $\mathcal{A}_1$ and those of $\mathcal{A}_2$.

Definition 4. *The binary relation $\sqsubseteq$ on types is the largest relation that satisfies the following conditions: (i) If $\tau \sqsubseteq q$, then $\tau = q$. (ii) If $\tau \sqsubseteq \bigwedge_{j<\beta} \sigma_j \to \sigma'$, then $\tau = \bigwedge_{i<\alpha} \tau_i \to \tau'$, with $\tau' \sqsubseteq \sigma'$ and $\forall i < \alpha. \exists j < \beta. \tau_i \sqsubseteq \sigma_j$. Given a homomorphism $f : \mathcal{A}_1 \to \mathcal{A}_2$, we define the refinement relation $\tau_1 \sqsubseteq_f \tau_2$ by $f(\tau_1) \sqsubseteq \tau_2$.*

The refinement relation can be considered as a generalization of the kinding relation $\tau :: \kappa$ (which means "a type τ has sort κ") used in Kobayashi and Ong's type systems [3, 5]. Note that the relation $\sqsubseteq$ is *not* a subtyping relation; the type constructor $\to$ is *covariant* wrt $\sqsubseteq$.

Example 2. $\sqsubseteq$ is reflexive and transitive. If $q_1 \neq q_2$, $q_1 \to q \sqsubseteq (q_1 \wedge q_2) \to q$ holds but $(q_1 \wedge q_2) \to q \sqsubseteq q_1 \to q$ does not. If $f(q_1) = f(q_2) = q$, then $(q_1 \wedge q_2) \to q_1 \sqsubseteq_f q \to q$. □

We write $\Gamma_1 \sqsubseteq \Gamma_2$ if for each $F : \tau \in \Gamma_1$, there exists $F : \sigma \in \Gamma_2$ such that $\tau \sqsubseteq \sigma$. The definition of $\Gamma_1 \sqsubseteq_f \Gamma_2$ is similar. The following is a key lemma to obtain the main result.

Lemma 5. *Let $\mathcal{A}$ be a* deterministic *trivial automaton and $\mathcal{G}$ be a recursion scheme. If $\mathcal{G}$ is typable in $\mathcal{T}_{\mathcal{A}}$, then $\Gamma_{\mathcal{G}}^r$ (where r is a run tree of $\mathcal{A}$ over $[\![\mathcal{G}]\!]$, which is unique by the assumption that $\mathcal{A}$ is deterministic) is the minimum type environment in $\{\Gamma \mid \vdash_{\mathcal{A}} \mathcal{G} : \Gamma\}$ with respect to $\sqsubseteq$.*

Proof Sketch Assume $\vdash_{\mathcal{A}} \mathcal{G} : \Gamma$. By the proof of Lemma 1 (which is constructive in the sense that it gives a procedure to construct a derivation tree for t' from that of t), we can construct a type derivation for each term occurring in $\mathcal{T}^\infty$ (recall that $\mathcal{T}^\infty$ is a tree representing an infinite reduction sequence). For $\mathcal{T}^\infty(p) = (p', uv_n \dots v_1))$, let $\phi_{(p,n)}$ be the type assigned to the term u. Then, we can prove by co-induction that $\theta_{(\rightarrow,p,n)} \sqsubseteq \phi_{(p,n)}$ holds for every p and n. Let $\mathcal{T}^{-1}(F) = \{(p,n) \mid \mathcal{T}^\infty(p) = (p', Fu_n \dots u_1)\}$. Thus, we have:

$$\Gamma^r_{\mathcal{G}} = \{F : \theta_{(\rightarrow,p,n)} \mid (p,n) \in \mathcal{T}^{-1}(F)\} \sqsubseteq \{F : \phi_{(p,n)} \mid (p,n) \in \mathcal{T}^{-1}(F)\} \subseteq \Gamma$$

as required. □

The following main result implies that if $\Gamma \vdash_{\mathcal{A}_2} \mathcal{G}$, then it suffices to consider only refinements of Γ to check whether $\mathcal{G}$ is well-typed in $\mathcal{T}_{\mathcal{A}_1}$.

Theorem 5. *Let $\mathcal{A}_1$ be a trivial automaton, $\mathcal{A}_2$ be a deterministic trivial automaton, $\mathcal{G}$ be a recursion scheme and $f : \mathcal{A}_1 \rightarrow \mathcal{A}_2$ be a homomorphism of automata. Assume $\Gamma \vdash_{\mathcal{A}_2} \mathcal{G}$. Then, $\mathcal{G}$ is typable in $\mathcal{T}_{\mathcal{A}_1}$ iff $\mathcal{G}$ is so under some type environment Γ' such that $\Gamma' \sqsubseteq_f \Gamma$.*

Proof. The "if" direction is trivial. To prove the converse, assume $\mathcal{G}$ is well-typed in $\mathcal{T}_{\mathcal{A}_1}$. By the soundness of the type system, $[\![\mathcal{G}]\!] \in \mathcal{L}_{\mathcal{A}_1}$ holds, i.e. there exists a run tree r_1 of $\mathcal{A}_1$ over $[\![\mathcal{G}]\!]$. Let r_2 be a unique run tree of $\mathcal{A}_2$ over $[\![\mathcal{G}]\!]$. It is easy to show that $r_2(p) = f(r_1(p))$. Then, by the definition of $\Gamma^{r_1}_{\mathcal{G}}$ and $\Gamma^{r_2}_{\mathcal{G}}$, $f(\Gamma^{r_1}_{\mathcal{G}}) \sqsubseteq \Gamma^{r_2}_{\mathcal{G}}$. By Lemma 5, for any type environment Γ such that $\Gamma \vdash_{\mathcal{A}_2} \mathcal{G}$, we have $f(\Gamma^{r_1}_{\mathcal{G}}) \sqsubseteq \Gamma^{r_2}_{\mathcal{G}} \sqsubseteq \Gamma$, which implies $\Gamma^{r_1}_{\mathcal{G}} \sqsubseteq_f \Gamma$. □

As the sort system is equivalent to the type system for the automaton $\mathcal{A}^\top$ and there are only finitely many refinements of a finite sort, we obtain:

Corollary 1. *If $\mathcal{G}$ is a finitely sorted recursion scheme (i.e. the sort of every non-terminal in $\mathcal{G}$ is finite), then it is decidable whether $[\![\mathcal{G}]\!]$ is accepted by $\mathcal{A}$.*

We can obtain a model checking algorithm by modifying Kobayashi's algorithm [9] for simply-typed recursion schemes.

6 Applications

We now discuss how the foregoing theory can be applied to verification of functional programs written in languages with advanced type systems (like polymorphism and recursive types). As shown in [3, 4], a higher-order functional program can be easily transformed into a recursion scheme that simulates the output or the event sequences of the source program, and then model-checked. Since the recursion scheme thus obtained is well-sorted under a similar type system, we focus here on model-checking of recursion schemes that are well-sorted under advanced type systems.

As already mentioned, polymorphic types and recursive types may be regarded as restricted forms of infinite intersection types (or sorts). For example, a (predicative) polymorphic type $\forall\alpha.\tau$ can be regarded as $\bigwedge\{[\sigma/\alpha]\tau \mid$

σ is a *finite* sort$\}$ with infinite width, and a recursive sort $\mu X.o \to X$ as a sort $o \to o \to \ldots$ with an infinite path. Thus, the model-checking problem of interest is: "Given a recursion scheme $\mathcal{G}$ well-sorted under a certain fragment $\mathcal{S}$ of the sort system of Section 3 and an automaton $\mathcal{A}$, is $[\![\mathcal{G}]\!]$ accepted by $\mathcal{A}$?" We discuss below decidable and undecidable fragments of the sort system.

Decidable fragments By Corollary 1, if $\mathcal{S}$ allows only finite sorts (intersection sorts with finite width and depth), then the model-checking problem is decidable. The ML-style let-polymorphism (where the set of rewriting rules is of the form $F_1\ \widetilde{x} \to t_1; \cdots; F_m\ \widetilde{x} \to t_m; S \to t_{m+1}$ and F_i can have polymorphic sorts only in $t_j (j > i)$) satisfies this condition. It is easy to see that if $F_1\ \widetilde{x} \to t_1; \cdots; F_m\ \widetilde{x} \to t_m; S \to t_{m+1}$ is well-typed under the ML-style polymorphic type system, then the recursion scheme is well-sorted by using only finite sorts.

Another fragment that has only finite sorts is the system **S** studied in [10]. As system **S** contains rank-2 intersection with polymorphic recursion ($\mathbf{I_2}$ + **REC-INT**, for which the typability is undecidable [11]) as a subsystem, this is an interesting example for which well-sorting is undecidable, but the model-checking problem for well-sorted recursion schemes is decidable. We do not know whether the model-checking problem is decidable for the fragment with Milner-Mycroft-style polymorphic recursion.

Remark 3. Note that if a recursion scheme is well-sorted by using finite sorts, then it can be transformed into an equivalent, simply-typed recursion scheme. (To see why, observe that a function of sort $\tau_1 \wedge \tau_2 \to \tau$ can be transformed into a function of sort $\tau_1 \to \tau_2 \to \tau$). Thus, extending simply-typed recursion schemes to those with finite intersection sorts does not increase the expressive power of recursion schemes. Our approach of using intersection sorts would, however, be more efficient, as the transformation from a finitely-sorted recursion scheme into a simply-typed recursion scheme will blow up the size of the recursion scheme.

Undecidable Fragments The model-checking problem is undecidable for fragments that contain either recursive types or System F-style polymorphic types. With System F-style (impredicative) polymorphism, we can encode a natural number as a term of type $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$ and express the successor, predecessor, and zero-equality test on natural numbers. Thus, by a combination with recursion,[4] we can encode a Minsky machine [8] M into a recursion scheme $\mathcal{G}$ such that M halts if and only if $[\![\mathcal{G}]\!]$ contains a terminal `h`. A similar reasoning applies to recursion schemes with recursive types.

For these undecidable fragments, we can still obtain a sound but incomplete model-checking algorithm, by restricting the refinement. For example, for recursive sorts, we can restrict their refinements to recursive types (i.e. regular infinite intersection types) of a fixed size, so that the number of possible refinements for

[4] Note that although terms of System F are strongly normalizing, we have recursion as a primitive.

each recursive sort is finite. Then Theorem 5 can be used to obtain a sound model-checking algorithm.

7 Related Work

The model checking of recursion schemes has been studied extensively [1, 2, 5, 12, 13], and applied to higher-order program verification [3, 4, 9]. Knapik et al. [1] showed the decidability of the modal μ-calculus model-checking of *safe* recursion schemes, and Ong [2] showed the decidability for arbitrary (typed) recursion schemes. Kobayashi and Ong [3, 5] recently proposed type-based model checking algorithms for recursion schemes. To our knowledge, all the previous studies dealt with *simply-typed* recursion schemes (simply-typedness was a part of the definition of recursion schemes).

Infinite intersection types have been studied by Leivant [7] and Bonsangue and Kok [14]. One of the main advantages of their type systems is that infinite intersection types give a "natural master formalism" [7] for various type disciplines. It is for this reason that we have introduced infinite intersection types for recursion schemes. To our knowledge, the previous type systems [7, 14] do not allow types having infinite paths (like the type τ defined by $\tau((11)^*) = \rightarrow$, $\tau((11)^*1) = \wedge^1$, and $\tau((11)^*2) = \mathtt{o}$).

There may be some connection between our work and studies on infinitely λ-calculi [15–17], where infinite objects generated by infinite reductions of λ-terms are considered. Note that the model checking of recursion schemes is also concerned about properties of the infinite objects generated by λ-terms. Direct connection is however unclear, since different properties of the infinite objects are considered. Tatsuta [16] showed that there is no decidable type system that characterizes the class of hereditary head-normalizing terms (λ-terms whose Böhm trees do not have $\perp$), and also gave a type-based characterization of that class by using an intersection type system with a countably infinite set of types.

8 Conclusion

We have developed an infinite intersection type system that is equivalent to the model-checking of untyped recursion schemes for safety properties (the properties expressed by trivial automata). Future work includes an extension of the type system to deal with the full modal μ-calculus, along the line of Kobayashi and Ong's work for typed recursion schemes [5]. It is also left for future work to find good decidable restrictions of the infinite intersection type system and apply them to verification of programs written in a language with polymorphic and/or recursive types.

Acknowledgments We would like to thank anonymous referees for useful comments. This work was partially supported by Kakenhi 20240001.

References

1. Knapik, T., Niwinski, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: FoSSaCS 2002. Volume 2303 of Lecture Notes in Computer Science., Springer-Verlag (2002) 205–222
2. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006, IEEE Computer Society Press (2006) 81–90
3. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages. (2009) 416–428
4. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: POPL. (2010) To appear.
5. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009, IEEE Computer Society Press (2009) 179–188
6. van Bakel, S.: Intersection type assignment systems. Theor. Comput. Sci. **151**(2) (1995) 385–435
7. Leivant, D.: Discrete polymorphism. In: LISP and Functional Programming. (1990) 288–297
8. Minsky, M.L.: Computation: Finite and infinite Machines. Prentice-Hall (1967)
9. Kobayashi, N.: Model-checking higher-order functions. In: Proceedings of PPDP 2009, ACM Press (2009)
10. Hallett, J.J., Kfoury, A.J.: Programming examples needing polymorphic recursion. Electr. Notes Theor. Comput. Sci. **136** (2005) 57–102
11. Terauchi, T., Aiken, A.: On typability for rank-2 intersection types with polymorphic recursion. In: LICS, IEEE Computer Society (2006) 111–122
12. Knapik, T., Niwinski, D., Urzyczyn, P.: Deciding monadic theories of hyperalgebraic trees. In: TLCA 2001. Volume 2044 of Lecture Notes in Computer Science., Springer-Verlag (2001) 253–267
13. Aehlig, K., de Miranda, J.G., Ong, C.H.L.: The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In: TLCA 2005. Volume 3461 of Lecture Notes in Computer Science., Springer-Verlag (2005) 39–54
14. Bonsangue, M.M., Kok, J.N.: Infinite intersection types. Inf. Comput. **186**(2) (2003) 285–318
15. Berarducci, A., Dezani-Ciancaglini, M.: Infinite lambda-calculus and types. Theor. Comput. Sci. **212**(1-2) (1999) 29–75
16. Tatsuta, M.: Types for hereditary head normalizing terms. In Garrigue, J., Hermenegildo, M.V., eds.: FLOPS. Volume 4989 of Lecture Notes in Computer Science., Springer (2008) 195–209
17. Tatsuta, M.: Types for hereditary permutators. In: LICS, IEEE Computer Society (2008) 83–92