Complexity of Model-Checking Call-by-Value Programs

Takeshi Tsukada[1,2] and Naoki Kobayashi[3]

[1] University of Oxford
[2] JSPS Postdoctoral Fellow for Research Abroad
[3] The University of Tokyo

Abstract. This paper studies the complexity of the reachability problem (a typical and practically important instance of the model-checking problem) for simply-typed call-by-value programs with recursion, Boolean values, and non-deterministic branch, and proves the following results. (1) The reachability problem for order-3 programs is nonelementary. Thus, unlike in the call-by-name case, the order of the input program does not serve as a good measure of the complexity. (2) Instead, the *depth* of types is an appropriate measure: the reachability problem for depth-n programs is n-EXPTIME complete. In particular, the previous upper bound given by the CPS translation is not tight. The algorithm used to prove the upper bound result is based on a novel intersection type system, which we believe is of independent interest.

1 Introduction

A promising approach to verifying higher-order functional programs is to use higher-order model checking [7,8,15], which is a decision problem about the trees generated by higher-order recursion schemes. Various verification problems such as the reachability problem and the resource usage verification [5] are reducible to the higher-order model checking [8].

This paper addresses a variant of the higher-order model checking, namely, the reachability problem for simply-typed *call-by-value* Boolean programs. It is the problem to decide, given a program with Boolean primitives and a special constant meaning the failure, whether the evaluation of the program fails. This is a practically important problem that can be a basis for verification of programs written in call-by-value languages such as ML and OCaml. In fact, MoCHi [11], a software model-checker for a subset of OCaml, reduces a verification problem to a reachability problem for a call-by-value Boolean program.

In the previous approach [11], the rechability problem for call-by-value programs was reduced to that for call-by-name programs via the CPS transformation. From a complexity theoretic point of view, however, this reduction via the CPS transformation has a bad effect: the order of a function is raised by 2 for each increase of the arity of the function. Since the reachability of order-n call-by-name programs is $(n-1)$-EXPTIME complete in general, the approach may suffer from double exponential blow-up of the time complexity for each increase of the largest arity in a program. Thus, important questions are: Is the double exponential blow-up of the time complexity (with respect to the arity increase) inevitable? If not, what is the exact complexity of the reachability problem for call-by-value programs, and how can we achieve the exact complexity?

The above questions are answered in this paper. We first show that the *single* exponential blow-up with respect to the arity increase is inevitable for programs of order-3 or higher. This implies that when the arity is not fixed, the reachability problem for order-3 call-by-value programs is nonelementary. The key observation used in the proof is that the subset of natural numbers $\{0, 1, \dots, \mathbf{exp}_n(2) - 1\}$ (here $\mathbf{exp}_n(k)$ is the nth iterated exponential function, defined by $\mathbf{exp}_0(k) = k$ and $\mathbf{exp}_{n+1}(k) = 2^{\mathbf{exp}_n(k)}$) can be embedded into the set of values of the type $\overbrace{\mathbb{B} \to \mathbb{B} \to \cdots \to \mathbb{B}}^{n} \to \mathbb{B}$ by using non-determinism.

Second, we show the *depth* of types is an appropriate measure, i.e. the reachability problem for depth-n programs is n-EXPTIME complete. The depth of function type is defined by $\mathit{depth}(\kappa \to \kappa') = \max\{\mathit{depth}(\kappa) + 1, \mathit{depth}(\kappa') + 1\}$. In particular, the previous bound given by the CPS translation is not tight. To prove the upper-bound, we develop a novel intersection type system that completely characterises programs that reach the failure. Since the target is a call-by-value language with effects (i.e. divergence, non-determinism and failure), the proposed type system is much different from that for call-by-name calculi [18,7,9], which we believe is of the independent interest.

Organisation of the paper Section 2 defines the problem addressed in the paper. Section 3 proves that the reachability problem for order-3 programs is nonelementary. Section 4 provides a sketch of the proof of n-EXPTIME hardness of the reachability problem for depth-n programs. In Section 5, we develop an intersection type system that characterises the reachability problem, and a type-checking algorithm. We discuss related work in Section 6 and conclude in Section 7. For the space limitation, we omit some details and proofs, which are found in a long version available from the first author's web page.

2 Call-by-Value Reachability Problem

The target language of the paper is a simply-typed call-by-value calculus with recursion, product types (restricted to argument positions), Boolean and non-deterministic branch. Simple types are called *sorts* in order to avoid confusion with intersection types introduced later. The sets of *sorts*, *terms* and *function definitions* (*definitions* for short) are defined by the following grammar:

$$\begin{array}{llcl}
(\mathit{Sorts}) & \kappa, \iota & ::= & \mathbb{B} \mid \kappa_1 \times \ldots \times \kappa_n \to \iota \\
(\mathit{Terms}) & s, t, u & ::= & x \mid f \mid \lambda\langle x_1, \dots, x_n\rangle.t \mid t\ \langle u_1, \dots, u_n\rangle \\
 & & \mid & t \oplus u \mid \mathtt{t} \mid \mathtt{f} \mid \mathbf{if}(t, u_1, u_2) \mid \mathfrak{F}_\kappa \mid \Omega_\kappa \\
(\mathit{Definitions}) & D & ::= & \{f_i = \lambda\langle x_{i,1}, \dots, x_{i,n_i}\rangle.t_i\}_{i \le m},
\end{array}$$

where $\langle x_1, \dots, x_n\rangle$ (resp. $\langle u_1, \dots, u_n\rangle$) is a non-empty sequence of variables (resp. terms). The sort $\mathbb{B}$ is for Boolean values and the sort $\kappa_1 \times \ldots \times \kappa_n \to \iota$ is for

$$\frac{b \in \{\mathtt{t}, \mathtt{f}\}}{\Delta \mid \mathcal{K} \vdash b :: \mathbb{B}} \qquad \frac{x :: \kappa \in \mathcal{K}}{\Delta \mid \mathcal{K} \vdash x :: \kappa} \qquad \frac{f :: \kappa \in \Delta}{\Delta \mid \mathcal{K} \vdash f :: \kappa} \qquad \frac{\Delta \mid \mathcal{K}, \vec{x} :: \vec{\kappa} \vdash t :: \iota}{\Delta \mid \mathcal{K} \vdash \lambda\langle\vec{x}\rangle.t :: \vec{\kappa} \to \iota}$$

$$\frac{\Delta \mid \mathcal{K} \vdash t :: \vec{\kappa} \to \iota' \qquad \Delta \mid \mathcal{K} \vdash \vec{u} :: \vec{\kappa}}{\Delta \mid \mathcal{K} \vdash t\ \langle\vec{u}\rangle :: \iota} \qquad \frac{\Delta \mid \mathcal{K} \vdash t :: \mathbb{B} \quad \Delta \mid \mathcal{K} \vdash u_i :: \kappa\ (i \in \{1,2\})}{\Delta \mid \mathcal{K} \vdash \mathbf{if}(t, u_1, u_2) :: \kappa}$$

$$\frac{\Delta \mid \mathcal{K} \vdash t :: \kappa \qquad \Delta \mid \mathcal{K} \vdash u :: \kappa}{\Delta \mid \mathcal{K} \vdash t \oplus u :: \kappa} \qquad \frac{}{\Delta \mid \mathcal{K} \vdash \mathfrak{F}_\kappa :: \kappa} \qquad \frac{}{\Delta \mid \mathcal{K} \vdash \Omega_\kappa :: \kappa}$$

Fig. 1. Sorting rules for terms

functions that take an n-tuple as the argument and returns a value of ι. A term is a variable x, a function symbol f (that is a variable expected to be defined in D), an abstraction $\lambda\langle x_1, \dots, x_n\rangle.t$ that takes an n-tuple as its argument, an application $t\ \langle u_1, \dots, u_n\rangle$ of t to n-tuple $\langle u_1, \dots, u_n\rangle$, a non-deterministic branch $t_1 \oplus t_2$, a truth value ($\mathtt{t}$ or $\mathtt{f}$), a conditional branch $\mathbf{if}(t, u_1, u_2)$, a special constant $\mathfrak{F}_\kappa$ (standing for 'Fail') to which the reachability is considered, or divergence Ω_κ. A function definition is a finite set of elements of the form $f = \lambda\langle x_1, \dots, x_n\rangle.t$, which defines functions by mutual recursion. If $(f = \lambda\langle\vec{x}\rangle.t) \in D$, we write $D(f) = \lambda\langle\vec{x}\rangle.t$. The domain $\mathrm{dom}(D)$ of D is $\{f \mid (f = \lambda\langle\vec{x}\rangle.t) \in D\}$.

For notational convenience, we use the following abbreviations. We write $\vec{x}$ for a *non-empty* sequence of variables $x_1, \dots, x_n$, and simply write $\lambda\langle x_1, \dots, x_n\rangle.t$ as $\lambda\langle\vec{x}\rangle.t$. Similarly, $t\ \langle u_1, \dots, u_n\rangle$ is written as $t\ \langle\vec{u}\rangle$, where $\vec{u}$ indicates the sequence $u_1, \dots, u_n$, and $\kappa_1 \times \dots \times \kappa_n \to \iota$ as $\vec{\kappa} \to \iota$, where $\vec{\kappa} = \kappa_1, \dots, \kappa_n$. Note that $\vec{\kappa} \to \iota$ is *not* $\kappa_1 \to \dots \to \kappa_n \to \iota$. Sort annotation of $\mathfrak{F}_\kappa$ and Ω_κ are often omitted. For a 1-tuple $\langle t\rangle$, we often write just t.

The sort system is defined straightforwardly. A *sort environment* is a finite set of sort bindings of the form $x :: \kappa$ (here a double-colon is used for sort bindings and judgements, in order to distinguish them from type bindings and judgements). We write $\mathcal{K}(x) = \kappa$ if $x :: \kappa \in \mathcal{K}$. A *sort judgement* is of the form $\Delta \mid \mathcal{K} \vdash t :: \kappa$, where Δ is the sort environment for function symbols and $\mathcal{K}$ is the sort environment for free variables of t. Given sequences $\vec{x}$ and $\vec{\kappa}$ of the same length, we write $\vec{x} :: \vec{\kappa}$ for $x_1 :: \kappa_1, \dots, x_n :: \kappa_n$. Given sequences $\vec{t}$ and $\vec{\kappa}$ of the same length, we write $\Delta \mid \mathcal{K} \vdash \vec{t} :: \vec{\kappa}$ just if we have $\Delta \mid \mathcal{K} \vdash t_i :: \kappa_i$ for all $i \leq n$, where n is the length of $\vec{t}$. The sorting rules are listed in Fig. 1.

When term t does not contain function symbols, we simply write $\emptyset \mid \mathcal{K} \vdash t :: \kappa$ as $\mathcal{K} \vdash t :: \kappa$. We assume that terms in the sequel are explicitly typed, i.e. every term is equipped with a sort derivation for it and we can freely refer to sorts of subterms and variables in the term. For function definitions, a judgement is of the form $\vdash D :: \Delta$, which is derived by the following rule:

$$\frac{\Delta \mid \emptyset \vdash D(f) :: \kappa \qquad (\text{for every } f :: \kappa \in \Delta)}{\vdash D :: \Delta}$$

A *program* is a pair of a definition D and a term t of the ground sort $\mathbb{B}$ with $\vdash D :: \Delta$ and $\Delta \mid \emptyset \vdash t :: \mathbb{B}$ for some Δ. A program is written as `let rec` D `in` t. A program `let rec` $\emptyset$ `in` t with no function symbols is simply written as t.

The set of *values* is defined by: $v, w ::= \lambda\langle\vec{x}\rangle.t \mid \mathtt{t} \mid \mathtt{f}$. Recall that $\vec{x}$ is a non-empty sequence. *Evaluation contexts* are defined by: $E ::= \Box \mid E\ \langle\vec{t}\rangle \mid v\ \langle w_1, \dots, w_{k-1}, E, t_{k+1}, \dots, t_n\rangle \mid \mathbf{if}(E, t_1, t_2)$. Therefore arguments are evaluated left-to-right. The reduction relation on terms is defined by the rules below:

$$\begin{array}{llll} E[(\lambda\langle\vec{x}\rangle.t)\ \langle\vec{v}\rangle] \longrightarrow E[\,[\vec{v}/\vec{x}]t\,] & \quad E[t_1 \oplus t_2] & \longrightarrow E[t_i] & (\text{for } i = 1, 2) \\ E[\mathbf{if}(\mathtt{t}, t_1, t_2)] \longrightarrow E[t_1] & \quad E[\mathbf{if}(\mathtt{f}, t_1, t_2)] & \longrightarrow E[t_2]. & \end{array}$$

We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. The reduction relation is not deterministic because of the non-deterministic branch. A closed well-typed term t cannot be reduced just if (1) t is a value, (2) $t = E[\mathfrak{F}]$ or (3) $t = E[\Omega]$. In the second case, t immediately fails and in the third case, t never fails since Ω diverges. So we do not need to consider further reduction steps for $E[\mathfrak{F}]$ and $E[\Omega]$. By this design choice, $\longrightarrow$ is terminating.

Lemma 1. *If $\emptyset \vdash t :: \kappa$, then t has no infinite reduction sequence.*

Given a function definition D, the reduction relation $\longrightarrow_D$ is defined by the same rules as $\longrightarrow$ and the following additional rule:

$$E[f] \longrightarrow_D E[D(f)].$$

We write $\longrightarrow_D^*$ for the reflexive and transitive closure of $\longrightarrow_D$. Note that reduction by $\longrightarrow_D$ does not terminate in general.

Definition 1 (Reachability Problem). We say a program $\mathtt{let\,rec}\ D\ \mathtt{in}\ t$ *fails* if $t \longrightarrow_D^* E[\mathfrak{F}]$ for some E. The *reachability problem* is the problem to decide whether a given program fails.

Example 1. Let $t_0 = \lambda f.\mathbf{if}(f\,\mathtt{t}, \mathbf{if}(f\,\mathtt{t}, \Omega, \mathfrak{F}), \Omega)$, which calls the argument f (at most) twice with the same argument $\mathtt{t}$ and fails just if the first call returns $\mathtt{t}$ and the second call $\mathtt{f}$. Let $u_0 = (\lambda x.\mathtt{t}) \oplus (\lambda x.\mathtt{f})$ and $e_1 = t_0\,u_0$. Then e_1 has just two reduction sequences starting from $e_1 \longrightarrow t_0\,(\lambda x.\mathtt{t})$ and $e_1 \longrightarrow t_0\,(\lambda x.\mathtt{f})$, both of which do not fail. In the call-by-name setting, however, e_1 would fail since

$$\begin{aligned} e_1 &\longrightarrow \mathbf{if}(u_0\,\mathtt{t}, \mathbf{if}(u_0\,\mathtt{t}, \Omega, \mathfrak{F}), \Omega) \longrightarrow \mathbf{if}((\lambda x.\mathtt{t})\,\mathtt{t}, \mathbf{if}(u_0\,\mathtt{t}, \Omega, \mathfrak{F}), \Omega) \\ &\longrightarrow^* \mathbf{if}(u_0\,\mathtt{t}, \Omega, \mathfrak{F}) \longrightarrow \mathbf{if}((\lambda x.\mathtt{f})\,\mathtt{t}, \Omega, \mathfrak{F}) \longrightarrow^* \mathfrak{F}. \end{aligned}$$

Consider the program $e_1' = t_0\,u_0'$ where $u_0' = \lambda x.(\mathtt{t} \oplus \mathtt{f})$, in which the non-deterministic branch is delayed by the abstraction. Then e_1' would fail both in call-by-name and in call-by-value.

Example 2. Consider the program $P_2 = \mathtt{let\,rec}\ D_2\ \mathtt{in}\ e_2$, where $D_2 = \{f = \lambda x.f\,x\}$ and $e_2 = (\lambda y.\mathfrak{F})\,(f\,\mathtt{t})$. Then P_2 never fails because

$$e_2 = (\lambda y.\mathfrak{F})\,(f\,\mathtt{t}) \longrightarrow_{D_2} (\lambda y.\mathfrak{F})\,((\lambda x.f\,x)\,\mathtt{t}) \longrightarrow_{D_2} (\lambda y.\mathfrak{F})\,(f\,\mathtt{t}) = e_2 \longrightarrow_{D_2} \cdots.$$

In the call-by-name case, however, P_2 would fail since $(\lambda x.\mathfrak{F})\,(f\,\mathtt{t}) \longrightarrow \mathfrak{F}$.

Example 3. Consider the program $e_3 = (\lambda x.\mathtt{t})\,\mathfrak{F}$. Then e_3 (immediately) fails because $e_3 = E[\mathfrak{F}]$, where $E = (\lambda x.\mathtt{t})\,\Box$. In contrast, e_3 would not fail in the call-by-name setting, in which E is not an evaluation context and $e_3 \longrightarrow \mathtt{t}$.

We give a technically convenient characterisation of the reachability problem. Let $\{f_1, \dots, f_n\}$ be the set of function symbols in D. The *mth approximation* of f_i, written F_i^m, is the term obtained by expanding the definition m times, as is formally defined below:

$$\begin{aligned} F_i^0 &= \lambda\langle x_1, \dots, x_k\rangle.\Omega_\iota && \text{(where } f_i :: \kappa_1 \times \dots \times \kappa_k \to \iota \in \Delta) \\ F_i^{m+1} &= [F_1^m/f_1, \dots, F_n^m/f_n](D(f_i)). \end{aligned}$$

The mth approximation of t is defined by: $[t]_D^m = [F_1^m/f_1, \dots, F_n^m/f_n]t$.

Lemma 2. *Let $P = \mathtt{let\ rec}\ D\ \mathtt{in}\ t$ be a program. Then $t \longrightarrow_D^* E[\mathfrak{F}]$ for some E if and only if $[t]_D^n \longrightarrow^* E'[\mathfrak{F}]$ for some n and E'.*

Size of terms and programs The size of sorts is inductively defined by $|\mathbb{B}| = 1$ and $|\kappa_1 \times \dots \times \kappa_n \to \iota| = 1 + |\iota| + \sum_{i=1}^n |\kappa_i|$. The size of sort environments is given by $|\mathcal{K}| = \sum_{x::\kappa\in\mathcal{K}} |\kappa|$. The size of a term is defined straightforwardly (e.g. $|x| = 1$ and $|t\,\langle u_1, \dots, u_n\rangle| = 1 + |t| + \sum_{i=1}^n |u_i|$) except for the abstraction $|\lambda\langle x_1, \dots, x_n\rangle.t| = 1 + |t| + \sum_{i=1}^n (1 + |\kappa_i|)$, where κ_i is the sort of x_i. Here a term t is considered to be explicitly sorted, and thus the size of annotated sorts should be added. For programs, $|\mathtt{let\ rec}\ D\ \mathtt{in}\ t| = |t| + \sum_{f\in\mathrm{dom}(D)} |D(f)|$.

Order and depth of programs. Order is a well-known measure that characterises complexity of the *call-by-name* reachability problem [10,15] (it is $(n-1)$-EXPTIME complete for order-n programs) and, as we shall see, depth characterises complexity in the call-by-value case. *Order* and *depth* of sorts are defined by:

$$\begin{aligned} &\mathit{order}(\mathbb{B}) = \mathit{depth}(\mathbb{B}) = 0 \\ &\mathit{order}(\vec{\kappa} \to \iota) = \max\{\mathit{order}(\iota), \quad \mathit{order}(\kappa_1)+1, \dots, \mathit{order}(\kappa_n)+1\} \\ &\mathit{depth}(\vec{\kappa} \to \iota) = \max\{\mathit{depth}(\iota)+1, \mathit{depth}(\kappa_1)+1, \dots, \mathit{depth}(\kappa_n)+1\} \end{aligned}$$

For a sort environment, $\mathit{depth}(\mathcal{K}) = \max\{\mathit{depth}(\kappa) \mid x :: \kappa \in \mathcal{K}\}$. Order and depth of judgements are defined by $\varphi(\Delta \mid \mathcal{K} \vdash t :: \kappa) = \varphi(\kappa)$, where $\varphi \in \{\mathit{order}, \mathit{depth}\}$. The order of a sort derivation is the maximal order of judgements in the derivation. The order of a sorted term t is the order of its sort derivation $\Delta \mid \mathcal{K} \vdash t :: \kappa$. The order of a program $\mathtt{let\ rec}\ D\ \mathtt{in}\ t$ is the maximal order of terms t and $D(f)$ ($f \in \mathrm{dom}(D)$). The depth of derivations, sorted terms and programs are defined similarly.

3 Order-3 Reachability is Nonelementary

This section proves the following theorem.

Theorem 1. *The reachability problem for order-3 programs is nonelementary.*

The key observation is that, for every n, the subset of natural numbers $\{0, 1, \ldots, \mathbf{exp}_n(2) - 1\}$ can be implemented by $\overbrace{\mathbb{B} \to \cdots \to \mathbb{B}}^{n} \to \mathbb{B}$ in a certain sense (see Definition 2). The non-determinism of the calculus is essential to the construction. Note that in the call-by-name case, the set of closed terms (modulo observational equivalence) of this sort can be bounded by 4^{4^n}, since

$$\overbrace{\mathbb{B} \to \cdots \to \mathbb{B}}^{n} \to \mathbb{B} \cong \overbrace{\mathbb{B} \times \cdots \times \mathbb{B}}^{n} \to \mathbb{B}.$$

The proof in this section can be sketched as follows. Let $L \subseteq \{0,1\}^*$ be a language in n-EXPSPACE. We can assume without loss of generality that there exists a Turing machine M that accepts L and runs in space $\mathbf{exp}_n(x)$ (here x is the size of the input). Given a word w, we reduce its acceptance by M to the reachability problem of a program (say $P_{M,w}$) of the call-by-value calculus in Section 2 extended to have natural numbers up to $N \geq \mathbf{exp}_n(x)$ (Lemma 3). The order of $P_{M,w}$ is independent from M and w: it is 3 when the order of the natural number type is defined to be 1. Recall that the natural numbers up to $\mathbf{exp}_{n+x}(2) \geq \mathbf{exp}_n(x)$ can be implemented by the order-1 sort $\overbrace{\mathbb{B} \to \cdots \to \mathbb{B}}^{n+x} \to \mathbb{B}$. By replacing natural numbers in $P_{M,w}$ with the implementation, the acceptance of w by M can be reduced to the reachability problem of an order-3 program without natural numbers.

3.1 Simulating Turing Machine by Program with Natural Numbers

First of all, we define programs with natural numbers up to N, which is an extension of the typed calculus presented in Section 2. The syntax of sorts and terms is given by:

$$\begin{array}{lll} (\textit{Sorts}) & \kappa, \iota & ::= \ \cdots \mid \mathbb{N} \\ (\textit{Terms}) & s, t, u & ::= \ \cdots \mid \mathtt{S} \mid \mathtt{P} \mid \mathtt{EQ} \mid \underline{0} \mid \underline{1} \mid \cdots \mid \underline{N-1} \end{array}$$

The extended calculus has an additional ground sort $\mathbb{N}$ for (bounded) natural numbers. Constants $\mathtt{S}$ and $\mathtt{P}$ are functions of sort $\mathbb{N} \to \mathbb{N}$ meaning the successor and the predecessor functions, respectively, and $\mathtt{EQ}$ is a constant of sort $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$ which checks if two arguments are equivalent. A constant $\underline{n}$ indicates the natural number n. The set of values is defined by: $v ::= \cdots \mid \mathtt{S} \mid \mathtt{P} \mid \mathtt{EQ} \mid \underline{n}$. Function definitions and evaluation contexts are given by the same syntax as in Section 2, but terms and values may contain natural numbers. The additional reduction rules are given by

$$\begin{array}{rll} E[\mathtt{S}\,\underline{n}] \longrightarrow_D E[\underline{n+1}] & & (\text{if } n+1 < N) \\ E[\mathtt{P}\,\underline{n}] \longrightarrow_D E[\underline{n-1}] & & (\text{if } n-1 \geq 0) \\ E[\mathtt{EQ}\,\langle \underline{n}, \underline{n} \rangle] \longrightarrow_D E[\mathtt{t}] & & \\ E[\mathtt{EQ}\,\langle \underline{n}, \underline{m} \rangle] \longrightarrow_D E[\mathtt{f}] & & (\text{if } n \neq m). \end{array}$$

Note that $E[\mathtt{S}\,\underline{N-1}]$ and $E[\mathtt{P}\,\underline{0}]$ get stuck. A *program with natural numbers up to N* is a pair of a function definition D and a term t of sort $\mathbb{B}$, written as $\mathtt{let\ rec}\ D\ \mathtt{in}\ t$. We assume that programs in the sequel do not contain constant numbers except for $\underline{0}$. The order of $\mathbb{N}$ is defined as 1.

Lemma 3. *Let $L \subseteq \{0,1\}^*$ be a language and M be a deterministic Turing machine accepting L that runs in space $\mathbf{exp}_n(x)$ for some n. Then, for every word $w \in \{0,1\}^*$ of length k and natural number $N \geq \mathbf{exp}_n(k)$, one can construct a program $P_{M,w}$ with natural numbers up to N such that $P_{M,w}$ fails if and only if $w \in L$. Furthermore $P_{M,w}$ is of order-3 and can be constructed in polynomial time with respect to k.*

Proof. Let M be a Turing machine with states Q and tape symbols Σ and w be a word of length k. We can assume without loss of generality that $Q = \{\mathtt{t},\mathtt{f}\}^q$ (that is, the set of all sequences of length q consisting of $\mathtt{t}$ and $\mathtt{f}$) and $\Sigma = \{\mathtt{t},\mathtt{f}\}^l$.

A configuration is expressed as a value of sort[1]

$$\mathit{Config} = \overbrace{\mathbb{B} \times \cdots \times \mathbb{B}}^{q} \times \overbrace{(\mathbb{N} \to \mathbb{B}) \times \cdots \times (\mathbb{N} \to \mathbb{B})}^{l} \times \mathbb{N},$$

where the first part represents the current state, the second part the tape and the third part the position of the tape head. The program $P_{M,w}$ has one recursive function *isAccepted* of sort $\mathit{Config} \to \mathbb{B}$. It checks if the current state is a final state and it fails if so. Otherwise it computes the next configuration and passes it to *isAccepted* itself. The body of the program generates the initial configuration determined by w and passes it to the function *isAccepted*.

Clearly we can construct $P_{M,w}$ in polynomial time with respect to k (the length of w) and the order of $P_{M,w}$ is 3. □

3.2 Implementing Natural Numbers

Let ν_n be the order-1 sort defined by $\nu_0 = \mathbb{B}$ and $\nu_{n+1} = \mathbb{B} \to \nu_n$. We shall show that natural numbers up to $\mathbf{exp}_n(2)$ can be implemented as values of ν_n.

Intuitive Explanation. We explain the intuition behind the construction by using the set-theoretic model. Let $\mathbf{N} = \{0,1,\ldots,N-1\}$. We explain the way to express the set $2^{\mathbf{N}} \cong \{0,1,\ldots,2^N-1\}$ as (a subset of) non-deterministic functions of $\mathbb{B} \to \mathbf{N}$, i.e. functions of $\mathbb{B} \to \mathcal{P}(\mathbf{N})$, where $\mathcal{P}(\mathbf{N})$ is the powerset of $\mathbf{N}$. The set $(\mathbb{B} \Rightarrow \mathbf{N}) \subseteq (\mathbb{B} \to \mathcal{P}(\mathbf{N}))$ is defined by:

$$(\mathbb{B} \Rightarrow \mathbf{N}) = \{f : \mathbb{B} \to \mathcal{P}(\mathbf{N}) \mid f(\mathtt{t}) \cup f(\mathtt{f}) = \mathbf{N} \text{ and } f(\mathtt{t}) \cap f(\mathtt{f}) = \emptyset\}.$$

In other words, $f \in \mathbb{B} \to \mathcal{P}(\mathbf{N})$ is in $\mathbb{B} \Rightarrow \mathbf{N}$ if and only if, for every $i \in \mathbf{N}$, exactly one of $i \in f(\mathtt{t})$ and $i \in f(\mathtt{f})$ holds. Hence a function $f : \mathbb{B} \Rightarrow \mathbf{N}$ determines a function of $\mathbf{N} \to \mathbb{B}$, say $\hat{f}$, defined by $\hat{f}(i) = b$ iff $i \in f(b)$ ($b \in \{\mathtt{t},\mathtt{f}\}$).

[1] Strictly speaking, it is not a sort in our syntax because products are restricted to argument positions. But there is no problem since occurrences of *Config* in the following construction are also restricted to argument positions.

There is a bijection between the set of functions $\mathbf{N} \to \mathbb{B}$ and the subset of natural numbers $\{0, 1, \ldots, 2^N - 1\}$, given by binary encoding, i.e. $(\hat{f} : \mathbf{N} \to \mathbb{B}) \mapsto \sum_{i<N, \hat{f}(i)=\mathtt{t}} 2^i$. For example, consider the case that $N = 4$ and $\mathbf{N} = \{0, 1, 2, 3\}$. Then 6 (= 0110 in binary) is represented by $\hat{f}_6$ such that $\hat{f}_6(0) = \hat{f}_6(3) = \mathtt{f}$ and $\hat{f}_6(1) = \hat{f}_6(2) = \mathtt{t}$. Therefore f_6 is given by $f_6(\mathtt{t}) = \{1, 2\}$ and $f_6(\mathtt{f}) = \{0, 3\}$.

Now let us consider the way to define operations such as the successor, predecessor and equality test. The key fact is that there is a term (say get) that computes $\hat{f}(i)$ for $f \in \mathbb{B} \Rightarrow \mathbf{N}$ and $i \in \mathbf{N}$, and there exists a term (say put) that computes $g \in \mathbb{B} \Rightarrow \mathbf{N}$ such that $\hat{g} = \hat{f}[i \mapsto b]$ for $f \in \mathbb{B} \Rightarrow \mathbf{N}$, $i \in \mathbf{N}$ and $b \in \{\mathtt{t}, \mathtt{f}\}$. They are given by the following informal equations:

$$\begin{aligned}\mathtt{get}\,\langle f, i\rangle &= \mathbf{if}(f\,\mathtt{t} = i,\ \mathtt{t},\ \Omega) \oplus \mathbf{if}(f\,\mathtt{f} = i,\ \mathtt{f},\ \Omega)\\ \mathtt{put}\,\langle f, i, b\rangle &= \lambda c^{\mathbb{B}}.\big(\mathbf{if}(b = c, i, \Omega) \oplus ((\lambda j.\mathbf{if}(i \neq j, j, \Omega))\,(f\,c))\big)\end{aligned}$$

where $f :: \mathbb{B} \to \mathbf{N}$ and $i, j :: \mathbf{N}$ and $b, c :: \mathbb{B}$. Note that put would be incorrect in the call-by-name setting. By using these functions, we can write operations like successor, predecessor and equality test for $\mathbb{B} \Rightarrow \mathbf{N}$. For example, the equality test eq can be defined by $\mathit{eq} = \lambda\langle f, g\rangle.e\,\langle f, g, N-1\rangle$, where e is given by the following recursive definition:

$$e\,\langle f, g, i\rangle = \mathbf{if}((\mathtt{get}\,\langle f, i\rangle) = (\mathtt{get}\,\langle g, i\rangle),\quad \mathbf{if}(i = 0,\ \mathtt{t},\ e\,\langle f, g, (i-1)\rangle),\quad \mathtt{f}).$$

Formal Development. We formally define the notion of implementations and show that replacement of natural numbers with its implementations preserves reachability.

Definition 2 (Implementation of Natural Numbers). Let $\mathbf{N}$ be the tuple $(N, D, \kappa, \{V_i\}_{i\in\{0,1,\ldots,N-1\}}, \mathbf{eq}, \mathbf{s}, \mathbf{p}, \mathbf{z}, \mathbf{max})$, where N is a natural number, D is a function definition, κ is a sort, $\{V_i\}_i$ is an indexed set of pairwise disjoint sets of closed values of sort κ, $\mathbf{eq}$ is a closed value of sort $\kappa \times \kappa \to \mathbb{B}$, $\mathbf{s}$ and $\mathbf{p}$ are closed values of sort $\kappa \to \kappa$, and $\mathbf{z}$ and $\mathbf{max}$ are closed values of sort κ. Here we consider terms *without* natural numbers. We say $\mathbf{N}$ is an *implementation of natural numbers up to* N just if the following conditions hold (here $V = \bigcup_i V_i$).

- For every $v, v' \in V$, evaluation of $\mathbf{eq}\,\langle v, v'\rangle$, $\mathbf{s}\,v$ and $\mathbf{p}\,v$ under D never fails.
- $\mathbf{z} \in V_0$ and $\mathbf{max} \in V_{N-1}$.
- For every $v \in V_n$ and $v' \in V_{n'}$, $\mathbf{eq}\,\langle v, v'\rangle \longrightarrow_D^* \mathtt{t}$ if and only if $n = n'$, and $\mathbf{eq}\,\langle v, v'\rangle \longrightarrow_D^* \mathtt{f}$ if and only if $n \neq n'$.
- For every $v \in V_n$, $\mathbf{s}\,v \longrightarrow_D^* v'$ implies $v' \in V_{n+1}$ and if $n + 1 < N$ then $\mathbf{s}\,v \longrightarrow_D^* v'$ for some $v' \in V_{n+1}$. Similarly, $\mathbf{p}\,v \longrightarrow_D^* v'$ implies $v' \in V_{n-1}$ and if $n \geq 1$ then $\mathbf{p}\,v \longrightarrow_D^* v'$ for some $v' \in V_{n-1}$.

The sort of $\mathbf{N}$ is κ and the order of $\mathbf{N}$ is that of κ.

Given an implementation $\mathbf{N}$ of natural numbers up to N and a term t with natural numbers up to N, we write $t^{\mathbf{N}}$ for the term without natural numbers obtained by replacing constants with values given by $\mathbf{N}$, e.g.,

$$\underline{0}^{\mathbf{N}} = \mathbf{z} \qquad \mathtt{S}^{\mathbf{N}} = \mathbf{s} \qquad (t\,u)^{\mathbf{N}} = t^{\mathbf{N}}\,u^{\mathbf{N}} \qquad (\lambda x.t)^{\mathbf{N}} = \lambda x.(t^{\mathbf{N}}).$$

Note that programs do not contain constant numbers except for $\underline{0}$ by definition. Given a function definition D, $D^{\mathbf{N}}$ can be defined straightforwardly. See the long version for the concrete definition.

Lemma 4. *Let* `let rec` D `in` t *be a program with natural numbers up to* N*, and* $\mathbf{N}$ *be an implementation of natural numbers up to* N*. Then* `let rec` D `in` t *fails if and only if* `let rec` $D^{\mathbf{N}}$ `in` $t^{\mathbf{N}}$ *fails.*

Given a natural number $n \geq 1$, we present an implementation of natural numbers up to $\mathbf{exp}_n(2)$ whose order is 1. By using the implementation to the program constructed in Lemma 3, the nonelementary result for the reachability problem for order-3 programs is established.

For every n, we shall define an implementation $\mathbf{N}(n)$ of natural numbers up to $\mathbf{exp}_n(2)$ by induction on n. As for the base case, the natural numbers up to $\mathbf{exp}_0(2) = 2$ (i.e. $\{0, 1\}$) can be naturally implemented by using $\mathbb{B}$. We call this implementation $\mathbf{N}(0)$. As for the induction step, assuming an implementation $\mathbf{N} = (N, D, \kappa, \{V_i\}_i, \mathbf{eq}, \mathbf{s}, \mathbf{p}, \mathbf{z}, \mathbf{max})$ of natural numbers up to N, it suffices to construct an implementation of natural numbers up to 2^N, say ${}^{\mathbb{B}}\mathbf{N} = (2^N, D \cup D', \mathbb{B} \to \kappa, \{V'_i\}_{i \in \{0,1,\ldots,2^N-1\}}, \mathbf{eq}', \mathbf{s}', \mathbf{p}', \mathbf{z}', \mathbf{max}')$.

- The additional function definition D' defines `get`, `put` and other auxiliary functions used to define $\mathbf{s}'$ and others. The definitions of `get` and `put` are:

$$\mathtt{get} = \lambda\langle x^{\mathbb{B}\to\kappa}, i^{\kappa}\rangle.\ \ \mathbf{if}(\mathbf{eq}\,\langle x\,\mathtt{t}, i\rangle,\ \mathtt{t},\ \Omega)\ \oplus\ \mathbf{if}(\mathbf{eq}\,\langle x\,\mathtt{f}, i\rangle,\ \mathtt{f},\ \Omega)$$
$$\mathtt{put} = \lambda\langle x^{\mathbb{B}\to\kappa}, i^{\kappa}, b^{\mathbb{B}}\rangle.\lambda c^{\mathbb{B}}.\ \big(\mathbf{if}(b = c, i, \Omega)\ \oplus\ ((\lambda j.\mathbf{if}(\mathbf{eq}\,\langle i, j\rangle, \Omega, j))\,(x\,c))\big)$$

- Let $m < 2^N$ and $b_{N-1}\ldots b_0$ be its binary representation. Then V'_m is the set of values v of sort $\mathbb{B} \to \kappa$ such that
  1. $b_i = 1$ iff $v\,\mathtt{t} \longrightarrow^* v'$ for some $v' \in V_i$,
  2. $b_i = 0$ iff $v\,\mathtt{f} \longrightarrow^* v'$ for some $v' \in V_i$, and
  3. $v\,\mathtt{t} \longrightarrow^* v'$ or $v\,\mathtt{f} \longrightarrow^* v'$ implies $v' \in \bigcup_{i\in\{0,\ldots,N-1\}} V_i$.

Here $x = y$ is the shorthand for $\mathbf{if}(x, \mathbf{if}(y, \mathtt{t}, \mathtt{f}), \mathbf{if}(y, \mathtt{f}, \mathtt{t}))$. For $n \geq 1$, we define $\mathbf{N}(n+1) = {}^{\mathbb{B}}(\mathbf{N}(n))$. See the long version for the concrete definition of ${}^{\mathbb{B}}\mathbf{N}$.

Lemma 5. $\mathbf{N}(n)$ *is an implementation of natural numbers up to* $\mathbf{exp}_n(2)$*. Furthermore, the sort, the function definition and the operations of* $\mathbf{N}(n)$ *can be constructed in time polynomial with respect to* n*.*

Proof (Theorem 1). The claim follows from Lemmas 3, 4 and 5. Note that (i) $\mathbf{exp}_n(x) \leq \mathbf{exp}_{n+x}(2)$, and (ii) given an order-n program with natural numbers up to $\mathbf{exp}_m(2)$, the replacement of natural number constants with $\mathbf{N}(m)$ can be done in time polynomial with respect to m and the size of the program, and the resulting program is of order n (provided that $n \geq 2$). □

4 Depth-n Reachability is n-EXPTIME Hard

In this section, we show a sketch of the proof of Theorem 2 below.

Theorem 2. *For every $n > 0$, the reachability problem for depth-n programs is n-EXPTIME hard.*

We reduce the emptiness problem of order-n alternating pushdown systems, which is known to be n-EXPTIME complete [4], to the reachability problem for depth-n programs. The basic idea originates from the work of Knapik et al. [6], which simulates a deterministic higher-order pushdown automaton by a safe higher-order grammar.

Since Knapik et al. [6] considered call-by-name grammars, we need to fill the gap between call-by-name and call-by-value. A problem arises when a divergent term that would not be evaluated in the call-by-name strategy appears in an argument position. We use the non-deterministic branch and the Boolean values to overcome the problem. Basically, by our reduction, every term of the ground sort is of the form $\mathtt{f} \oplus s$, and thus one can choose whether s is evaluated or not, by selecting one of the two possible reduction $\mathtt{f} \oplus s \longrightarrow \mathtt{f}$ and $\mathtt{f} \oplus s \longrightarrow s$. A detailed proof can be found in the long version.

5 Intersection-Type-Based Model-Checking Algorithm

We develop an intersection type system that completely characterises the reachability problem and give an upper bound of complexity of the reachability problem by solving the typability problem.

5.1 Types

The pre-types are given by the following grammar:

$$\begin{array}{lrcl} \textit{(Value Pre-types)} & \theta & ::= & \mathtt{t} \mid \mathtt{f} \mid \bigwedge_{i\in I}(\theta_{1,i}\times\ldots\times\theta_{n,i} \to \tau_i) \\ \textit{(Term Pre-types)} & \tau, \sigma & ::= & \theta \mid \mathfrak{F}_\kappa \end{array}$$

The index I of the intersection is a finite set. We allow I to be the empty set, and we also write $\bigwedge \emptyset$ for the type. The subscript κ of $\mathfrak{F}_\kappa$ is often omitted. We use infix notation for intersection, e.g. $(\theta_1 \to \tau_1) \wedge (\theta_2 \to \tau_2)$. The intersection connective is assumed to be associative, commutative and idempotent. Thus types $\bigwedge_{i\in I}(\theta_{1,i}\times\ldots\times\theta_{n,i} \to \tau_i)$ and $\bigwedge_{j\in J}(\theta'_{1,j}\times\ldots\times\theta'_{n,j} \to \tau'_{n,j})$ are equivalent if $\{(\theta_{1,i},\ldots,\theta_{n,i},\tau_i) \mid i \in I\}$ and $\{(\theta'_{1,j},\ldots,\theta'_{n,j},\tau'_j) \mid j \in J\}$ are equivalent sets.

Value pre-types are types for values and term pre-types are those for terms.

The value pre-type $\mathtt{t}$ is for the Boolean value $\mathtt{t}$ and $\mathtt{f}$ for the Boolean value $\mathtt{f}$. The last one is for abstractions. It can be understood as the intersection of function types of the form $\theta_1\times\ldots\times\theta_n \to \tau$. The judgement $\lambda\langle\vec{x}\rangle.t : \theta_1\times\ldots\times\theta_n \to \tau$ means that, for all values $v_i : \theta_i$ (for every $i \le n$), one has $[\vec{v}/\vec{x}]t : \tau$. For example, $\lambda x.x : \mathtt{t} \to \mathtt{t}$ and $\lambda x.x : \mathtt{f} \to \mathtt{f}$. The judgement $\lambda\langle\vec{x}\rangle.t : \bigwedge_{i\in I}(\theta_{1,i}\times\ldots\times\theta_{n,i} \to \tau_i)$ means that, for every $i \in I$, one has $\lambda\langle\vec{x}\rangle.t : \theta_{1,i}\times\ldots\times\theta_{n,i} \to \tau_i$. Therefore, $\lambda x.x : (\mathtt{t} \to \mathtt{t}) \wedge (\mathtt{f} \to \mathtt{f})$.

The term pre-type $\mathfrak{F}$ means failure, i.e. $t : \mathfrak{F}$ just if $t \longrightarrow^* E[\mathfrak{F}]$. The term pre-type θ is for terms that is reducible to a value of type θ, i.e. $t : \theta$ just if $t \longrightarrow^* v$ and $v : \theta$ for some v. For example, consider $u_0 = (\lambda x.\mathtt{t}) \oplus (\lambda x.\mathtt{f})$ and $u'_0 = \lambda x.(\mathtt{t} \oplus \mathtt{f})$ in Example 1. Then $u_0 : \mathtt{t} \to \mathtt{t}$ since $u_0 \longrightarrow \lambda x.\mathtt{t}$, and $u_0 : \mathtt{t} \to \mathtt{f}$ since $u_0 \longrightarrow \lambda x.\mathtt{f}$. It is worth noting that $t : \theta_1$ and $t : \theta_2$ does not imply $t : \theta_1 \wedge \theta_2$, e.g. u_0 does not have type $(\mathtt{t} \to \mathtt{t}) \wedge (\mathtt{t} \to \mathtt{f})$. In contrast, $u'_0 : (\mathtt{t} \to \mathtt{t}) \wedge (\mathtt{t} \to \mathtt{f})$. So the difference between u_0 and u'_0 is captured by types.

Given a sort κ, the relation $\tau :: \kappa$, read "τ is a refinement of κ," is inductively defined by the following rules:

$$\frac{}{\mathtt{t} :: \mathbb{B}} \qquad \frac{}{\mathtt{f} :: \mathbb{B}} \qquad \frac{}{\mathfrak{F}_\kappa :: \kappa} \qquad \frac{\theta_{k,i} :: \kappa_k \qquad \tau_i :: \iota \quad (\text{for all } i \in I,\ k \in \{1, \dots, n\})}{\bigwedge_{i \in I}(\theta_{1,i} \times \dots \times \theta_{n,i} \to \tau_i) :: \kappa_1 \times \dots \times \kappa_n \to \iota}$$

Note that intersection is allowed only for pre-types of the same sort. So a pre-type like $((\mathtt{t} \to \mathtt{t}) \to \mathtt{t}) \wedge (\mathtt{t} \to \mathtt{t})$ is not a refinement of any sort. A *type* is a value pre-type with its sort $\theta :: \kappa$ or a term pre-type with its sort $\tau :: \kappa$. A type is often simply written as θ or τ.

Let $\theta, \theta' :: \kappa$ be value types of the same sort. We define $\theta \wedge \theta'$ by:

$$\mathtt{t} \wedge \mathtt{t} = \mathtt{t} \qquad \mathtt{f} \wedge \mathtt{f} = \mathtt{f} \qquad (\bigwedge_{i \in I}(\vec{\theta}_i \to \tau_i)) \wedge (\bigwedge_{j \in J}(\vec{\theta}_j \to \tau_j)) = \bigwedge_{i \in I \cup J}(\vec{\theta}_i \to \tau_i)$$

and $\mathtt{t} \wedge \mathtt{f}$ and $\mathtt{f} \wedge \mathtt{t}$ are undefined.

5.2 Typing Rules

A *type environment* Γ is a finite set of type bindings of the form $x : \theta$ (here x is a variable or a function symbol). We write $\Gamma(x) = \theta$ if $x : \theta \in \Gamma$. We assume type bindings respect sorts, i.e. $x :: \kappa$ implies $\Gamma(x) :: \kappa$. A *type judgement* is of the form $\Gamma \vdash t : \tau$. The judgement intuitively means that, if each free variable x in t is bound to a value of type $\Gamma(x)$, then at least one possible evaluation of t results in a value of type τ. We abbreviate a sequence of judgements $\Gamma \vdash t_1 : \tau_1, \dots, \Gamma \vdash t_n : \tau_n$ as $\Gamma \vdash \vec{t} : \vec{\tau}$. The typing rules are listed in Fig. 2.

Here are some notes on typing rules. Rule (Abs) can be understood as the (standard) abstraction rule followed by the intersection introduction rule. Rule (App) can be understood as the intersection elimination rule followed by the (standard) application rule. Note that intersection is introduced by (Abs) rule and eliminated by (App) rule, which is the converse of the call-by-name case [7]. Rule (Var) is designed for ensuring weakening. Rule (App-$\mathfrak{F}$1) reflects the fact that, if $t \longrightarrow^* E[\mathfrak{F}]$, then $t\,\langle\vec{u}\rangle \longrightarrow^* E'[\mathfrak{F}]$ where $E' = E\,\langle\vec{u}\rangle$. Rule (App-$\mathfrak{F}$2) reflects the fact that, if $t \longrightarrow v_0$ and $u_i \longrightarrow^* v_i$ for $i < l$, then $t\,\langle u_1, \dots, u_{l-1}, u_l, u_{l+1}, \dots, u_n\rangle \longrightarrow^* v_0\,\langle v_1, \dots, v_{l-1}, E[\mathfrak{F}], u_{l+1}, \dots, u_n\rangle$. The premises $t : \theta_0$ and $u_i : \theta_i$ $(i < l)$ ensure may-convergence of their evaluation.

Typability of a program is defined by using the notion of the nth approximation (see Section 2 for the definition). Let $P = \mathtt{let\ rec}\ D\ \mathtt{in}\ t$ be a program. Thus t is a term of sort $\mathbb{B}$ with free occurrences of function symbols. We say the program P has type τ (written as $\vdash P : \tau$) just if $\vdash [t]^n_D : \tau$ for some n.

$$\frac{x : \theta \wedge \theta' \in \Gamma \text{ for some } \theta'}{\Gamma \vdash x : \theta} \;(\textsc{Var})$$

$$\frac{b \in \{\mathtt{t}, \mathtt{f}\}}{\Gamma \vdash b : b} \;(\textsc{Bool})$$

$$\frac{}{\Gamma \vdash \mathfrak{F} : \mathfrak{F}} \;(\mathfrak{F})$$

$$\frac{\Gamma, \vec{x} : \vec{\theta_i} \vdash t : \tau_i \text{ for all } i \in I}{\Gamma \vdash \lambda\langle\vec{x}\rangle.t : \bigwedge_{i\in I}(\vec{\theta_i} \to \tau_i)} \;(\textsc{Abs})$$

$$\frac{\Gamma \vdash t : \bigwedge_{i\in I}(\vec{\theta_i} \to \tau_i) \quad \Gamma \vdash \vec{u} : \vec{\theta_l} \quad l \in I}{\Gamma \vdash t\,\langle\vec{u}\rangle : \tau_l} \;(\textsc{App})$$

$$\frac{\Gamma \vdash t : \mathfrak{F}}{\Gamma \vdash t\,\langle\vec{u}\rangle : \mathfrak{F}} \;(\textsc{App-}\mathfrak{F}1)$$

$$\frac{\Gamma \vdash t : \theta_0 \quad \Gamma \vdash u_1 : \theta_1 \quad \cdots \quad \Gamma \vdash u_{l-1} : \theta_{l-1} \quad \Gamma \vdash u_l : \mathfrak{F}}{\Gamma \vdash t\,\langle\vec{u}\rangle : \mathfrak{F}} \;(\textsc{App-}\mathfrak{F}2)$$

$$\frac{\Gamma \vdash t : \mathtt{t} \qquad \Gamma \vdash s_1 : \tau}{\Gamma \vdash \mathbf{if}(t, s_1, s_2) : \tau} \;(\text{C-T})$$

$$\frac{\Gamma \vdash t : \mathtt{f} \qquad \Gamma \vdash s_2 : \tau}{\Gamma \vdash \mathbf{if}(t, s_1, s_2) : \tau} \;(\text{C-F})$$

$$\frac{\Gamma \vdash t : \mathfrak{F}}{\Gamma \vdash \mathbf{if}(t, s_1, s_2) : \mathfrak{F}} \;(\text{C-}\mathfrak{F})$$

$$\frac{\exists i \in \{1, 2\} \quad \Gamma \vdash t_i : \tau}{\Gamma \vdash t_1 \oplus t_2 : \tau} \;(\textsc{Br})$$

Fig. 2. Typing Rules

Soundness and completeness of the type system can be proved by using a standard technique for intersection type systems, except that Substitution and De-Substitution Lemmas are restricted to substitution of values and Subject Reduction and Expansion properties are restricted to call-by-value reductions. For more details, see the long version of the paper.

Theorem 3. *$\vdash P : \mathfrak{F}$ if and only if P fails.*

5.3 Type-Checking Algorithm and Upper Bound of Complexity

We provide an algorithm that decides the typability of a given depth-n program P in time $O(\mathbf{exp}_n(poly(|P|)))$ for some polynomial $poly$. Let $P = \mathtt{let\ rec}\ D\ \mathtt{in}\ t$ and suppose that $\vdash D :: \Delta$, $\Delta \vdash t :: \mathbb{B}$ and $\Delta = \{f_i :: \delta_i \mid i \in I\}$.

We define $\mathcal{T}(\kappa) = \{\tau \mid \tau :: \kappa\}$ and $\mathcal{T}(\Delta) = \{\Gamma \mid \Gamma :: \Delta\}$. For $\tau, \sigma \in \mathcal{T}(\kappa)$, we write $\tau \preceq \sigma$ just if $\tau = \sigma \wedge \sigma'$ for some σ'. The ordering for type environments is defined similarly. Let $\mathcal{F}_D$ be a function on $\mathcal{T}(\Delta)$, defined by:

$$\mathcal{F}_D(\Theta) = \left\{ f : \bigwedge\{\vec{\theta} \to \tau \mid \Theta, \vec{x} : \vec{\theta} \vdash t : \tau\} \;\middle|\; (f = \lambda\langle\vec{x}\rangle.t) \in D \right\}.$$

The algorithm to decide whether $\vdash \mathtt{let\ rec}\ D\ \mathtt{in}\ t : \mathfrak{F}$ is shown in Fig. 3.

Termination of the algorithm comes from monotonicity of $\mathcal{F}_D$ and finiteness of $\mathcal{T}(\Delta)$. Correctness is a consequence of the following lemma and the monotonicity of the approximation (i.e. if $[t]^m_D$ fails and $m \le m'$, then $[t]^{m'}_D$ fails).

```
1   : $\Theta_0 := \{f : \bigwedge\emptyset \mid f \in \mathrm{dom}(\Delta)\},\ \Theta_1 = \mathcal{F}_D(\Theta_0),\ i := 1$
2   : while $\Theta_i \neq \Theta_{i-1}$ do
2-1 :     $\Theta_{i+1} := \mathcal{F}_D(\Theta_i)$
2-2 :     $i := i + 1$
3   : if $\Theta_i \vdash t : \mathfrak{F}$ then yes else no
```

Fig. 3. Algorithm checking if $\vdash$ `let rec` D `in` $t : \mathfrak{F}$

Lemma 6. *Suppose $\Delta \mid \mathcal{K} \vdash t :: \mathbb{B}$. Then $\emptyset \vdash [t]_D^n : \tau$ if and only if $\Theta_n \vdash t : \tau$.*

We shall analyse the cost of the algorithm. For a set A, we write $\#A$ for the number of elements. The *height* of a poset A is the maximum length of strictly increasing chains in A.

Lemma 7. *Let κ be a sort of depth n. Then $\#\mathcal{T}(\kappa) \leq \mathbf{exp}_{n+1}(2|\kappa|)$ and the height of $\mathcal{T}(\kappa)$ is bounded by $\mathbf{exp}_n(2|\kappa|)$.*

Lemma 8. *Let $\Delta \mid \mathcal{K} \vdash t :: \kappa$ be a sorted term of depth n, and $\Theta :: \Delta$. Assume that depth$(\mathcal{K}) \leq n - 1$. Then $A_{\Theta,t} = \{(\Gamma, \tau) \in \mathcal{T}(\mathcal{K}) \times \mathcal{T}(\kappa) \mid \Theta, \Gamma \vdash t : \tau\}$ can be computed in time $O(\mathbf{exp}_n(poly(|t|)))$ for some polynomial poly.*

Proof. We can compute $A_{\Theta,t}$ by induction on t. An important case is that the sort κ is of depth n. In this case, there exists $B_{\Theta,t} \subseteq \mathcal{T}(\mathcal{K}) \times \mathcal{T}(\kappa)$ such that (1) $(\Gamma, \tau) \in A_{\Theta,t}$ if and only if $(\Gamma, \tau') \in B_{\Theta,t}$ for some $\tau' \preceq \tau$ and (2) for each Γ, the number of elements in $B_{\Theta,t} \upharpoonright \Gamma = \{\tau \mid (\Gamma, \tau) \in B_{\Theta,t}\}$ is bounded by $|t|$. See the long version for the proof of this claim. By using $B_{\Theta,t}$ as the representation of $A_{\Theta,t}$, $A_{\Theta,t}$ can be computed in the desired bound. For other cases, one can enumerate all the elements in $A_{\Theta,t}$, since $\#A_{\Theta,t} \leq \mathbf{exp}_n(2(|\mathcal{K}| + |\kappa|)) \leq \mathbf{exp}_n(2|t|)$ (here we assume w.l.o.g. that each variable in dom$(\mathcal{K})$ appears in t). $\square$

Theorem 4. *The reachability problem for depth-n programs is in n-EXPTIME.*

Proof. By Lemma 8, each iteration of loop 2 in Fig. 3 runs in n-EXPTIME. Since the height of $\mathcal{T}(\Delta)$ is bounded by $\mathbf{exp}_n(2|\Delta|)$, one needs at most $\mathbf{exp}_n(2|\Delta|)$ iterations for loop 2, and thus loop 2 runs in n-EXPTIME. Again by Lemma 8, step 3 can be computed in n-EXPTIME. Thus the algorithm in Fig. 3 runs in n-EXPTIME for depth-n programs. $\square$

6 Related Work

Higher-order model checking. Model-checking recursion schemes against modal μ-calculus (known as higher-order model checking) has been proved to be decidable by Ong [15], and applied to various verification problems of higher-order programs [7,11,12,17]. The higher-order model-checking problem is n-EXPTIME complete for order-n recursion schemes [15]. The reachability problem for *call-by-name* programs is an instance of the higher-order model checking, and $(n-1)$-EXPTIME complete for order-n programs [10].

Model-checking call-by-value programs via the CPS translation. The previous approach for model-checking call-by-value programs is based on the CPS translation. Our result implies that the upper bound given by the CPS translation is not tight. However this does not imply that the CPS translation followed by call-by-name model-checking is inefficient. It depends on the model-checking algorithm. For example, the naïve algorithm in [7] following the CPS translation takes more time than our algorithm, but we conjecture that HorSat [2] following the CPS translation meets the tight bound.

Sato et al. [16] employed the selective CPS translation [14] to avoid unnecessary growth of the order, using a type and effect system to capture effect-free fragments and then added continuation parameters to only effectful parts.

Intersection types for call-by-value calculi. Davies and Pfenning [3] studied an intersection type system for a call-by-value effectful calculus and pointed out that the value restriction on the intersection introduction rule is needed. In our type system, the intersection introduction rule is restricted immediately after the abstraction rule, which can be considered as a variant of the value restriction.

Similarly to the previous work on type-based approaches for higher-order model checking [7,8,9], our intersection type system is a variant of the Essential Type Assignment System in the sense of van Bakel [18], in which the typing rules are syntax directed. Our syntax of intersection types differs from the standard one for call-by-name calculi. Our syntax is inspired by the embedding of the call-by-value calculus into the linear lambda calculus [13], in which the call-by-value function type $A \to B$ is translated into $!(A \multimap B)$ (recall that function types in our intersection type system is $\bigwedge_i(\tau_i \to \sigma_i)$).

Zeilberger [19] proposed a principled design of the intersection type system based on the idea from focusing proofs [1]. Its connection to ours is currently unclear, mainly because of the difference of the target calculi.

Our type system is designed to be complete. This is a characteristic feature that the previous work for call-by-value calculi [3,19] does not have.

7 Conclusion

We have studied the complexity of the reachability problem for call-by-value programs, and proved the following results. First, the reachability problem for order-3 programs is non-elementary, and thus the order of the program does not serve as a good measure of the complexity, in contrast to the call-by-name case. Second, the reachability problem for depth-n programs is n-EXPTIME complete, which improves the previous upper bound given by the CPS translation.

For future work, we aim to (1) develop an efficient model-checker for call-by-value programs, using the type system proposed in the paper, and (2) study the relationship between intersection types and focused proofs [1,19].

Acknowledgement. This work is partially supported by JSPS KAKENHI Grant Number 23220001.

References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. J. Log. Comput. 2(3), 297–347 (1992)
2. Broadbent, C.H., Kobayashi, N.: Saturation-based model checking of higher-order recursion schemes. In: CSL 2013. LIPIcs, vol. 23, pp. 129–148, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013)
3. Davies, R., Pfenning, F.: Intersection types and computational effects. In: ICFP 2000, pp. 198–208. ACM (2000)
4. Engelfriet, J.: Iterated stack automata and complexity classes. Inf. Comput. 95(1), 21–75 (1991)
5. Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Program. Lang. Syst. 27(2), 264–313 (2005)
6. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)
7. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: POPL 2009, pp. 416–428. ACM (2009)
8. Kobayashi, N.: Model checking higher-order programs. J. ACM 60(3), 20 (2013)
9. Kobayashi, N., Ong, C.-H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: LICS 2009, pp. 179–188. IEEE Computer Society (2009)
10. Kobayashi, N., Ong, C.-H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. Logical Methods in Computer Science 7(4) (2011)
11. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: PLDI 2011, pp. 222–233. ACM (2011)
12. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: POPL 2010, pp. 495–508. ACM (2010)
13. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. Electr. Notes Theor. Comput. Sci. 1, 370–392 (1995)
14. Nielsen, L.R.: A selective cps transformation. Electr. Notes Theor. Comput. Sci. 45, 311–331 (2001)
15. Ong, C.-H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS, pp. 81–90. IEEE Computer Society (2006)
16. Sato, R., Unno, H., Kobayashi, N.: Towards a scalable software model checker for higher-order programs. In: PEPM 2013, pp. 53–62. ACM (2013)
17. Tobita, Y., Tsukada, T., Kobayashi, N.: Exact flow analysis by higher-order model checking. In: Schrijvers, T., Thiemann, P. (eds.) FLOPS 2012. LNCS, vol. 7294, pp. 275–289. Springer, Heidelberg (2012)
18. van Bakel, S.: Intersection type assignment systems. Theor. Comput. Sci. 151(2), 385–435 (1995)
19. Zeilberger, N.: Refinement types and computational duality. In: PLPV 2009, pp. 15–26. ACM (2009)