Higher-Order Model Checking in Direct Style

Taku Terao[1,2(✉)], Takeshi Tsukada[1], and Naoki Kobayashi[1]

[1] The University of Tokyo, Tokyo, Japan
terao1984@is.s.u-tokyo.ac.jp

[2] JSPS Research Fellow, Tokyo, Japan

Abstract. Higher-order model checking, or model checking of higher-order recursion schemes, has been recently applied to fully automated verification of functional programs. The previous approach has been *indirect*, in the sense that higher-order functional programs are first abstracted to (call-by-value) higher-order Boolean programs, and then further translated to higher-order recursion schemes (which are essentially call-by-*name* programs) and model checked. These multi-step transformations caused a number of problems such as code explosion. In this paper, we advocate a more *direct* approach, where higher-order Boolean programs are directly model checked, without transformation to higher-order recursion schemes. To this end, we develop a model checking algorithm for higher-order call-by-value Boolean programs, and prove its correctness. According to experiments, our prototype implementation outperforms the indirect method for large instances.

1 Introduction

Higher-order model checking [14], or model checking of higher-order recursion schemes (HORS), has recently been applied to automated verification of higher-order functional programs [9,11,12,15,17]. A HORS is a higher-order tree grammar for generating a (possibly infinite) tree, and higher-order model checking is concerned about whether the tree generated by a given HORS satisfies a given property. Although the worst-case complexity of higher-order model checking is huge (k-EXPTIME complete for order-k HORS [14]), practical algorithms for higher-order model checking have been developed [4,8,16,18], which do not always suffer from the k-EXPTIME bottleneck.

A typical approach for applying higher-order model checking to program verification [11] is as follows. As illustrated on the left-hand side of Fig. 1, a source program, which is a *call-by-value* higher-order functional program, is first abstracted to a call-by-value, higher-order *Boolean* functional program, using predicate abstraction. The Boolean functional program is further translated to a HORS, which is essentially a call-by-*name* higher-order functional program, and then model checked. We call this approach *indirect*, as it involves many steps of program transformations. This indirect approach has an advantage that, thanks to the CPS transformation used in the translation to HORS, various control

© Springer International Publishing AG 2016
A. Igarashi (Ed.): APLAS 2016, LNCS 10017, pp. 295–313, 2016.
DOI: 10.1007/978-3-319-47958-3_16

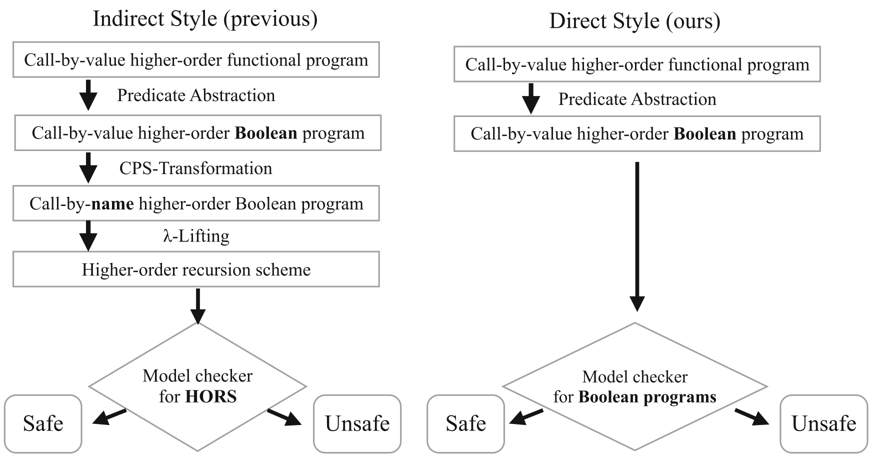
Indirect Style (previous)
Call-by-value higher-order functional program
Predicate Abstraction
Call-by-value higher-order **Boolean** program
CPS-Transformation
Call-by-**name** higher-order Boolean program
λ-Lifting
Higher-order recursion scheme
Model checker for **HORS**
Safe
Unsafe
Direct Style (ours)
Call-by-value higher-order functional program
Predicate Abstraction
Call-by-value higher-order **Boolean** program
Model checker for **Boolean programs**
Safe
Unsafe

Fig. 1. Overview: indirect vs. direct style

structures (such as exceptions and call/cc) and evaluation strategies (call-by-value and call-by-name) can be uniformly handled. The multi-step transformations, however, incur a number of drawbacks as well, such as code explosion and the increase of the order of programs (where the order of a program is the largest order of functions; a function is first-order if both the input and output are base values, and it is second-order if it can take a first-order function as an argument, etc.). The multi-step transformations also make it difficult to propagate the result of higher-order model checking back to the source program, e.g., for the purpose of counter-example-guided abstraction refinement (CEGAR), and certificate generation.

In view of the drawbacks of the indirect approach mentioned above, we advocate higher-order model checking in a more *direct* style, where call-by-value higher-order Boolean programs are directly model checked, without the translation to HORS, as illustrated on the right-hand side of Fig. 1. That would avoid the increase of the size and order of programs (recall that the complexity of higher-order model checking is k-EXPTIME complete for order-k HORS; thus the order is the most critical parameter for the complexity). In addition, the direct style approach would take an advantage of optimization using the control structure of the original program, which has been lost during the CPS-transformation in indirect style.

Our goal is then to develop an appropriate algorithm that directly solves the model-checking problem for call-by-value higher-order Boolean programs. We focus here on the reachability problem (of checking whether a given program reaches a certain program point); any safety properties can be reduced to the reachability problem in a standard manner.

From a purely theoretical point of view, this goal has been achieved by Tsukada and Kobayashi [20]. They developed an intersection type system for reachability checking of call-by-value higher-order Boolean programs, which gives a better (and exact in a certain sense) upper bound of the worst case complexity of the problem than the naïve indirect approach. However their algorithm, which basically enumerates all the types of subterms, is far from practical since the number of candidate types for a given subterm is hyper-exponential.

Now the challenge is to find an appropriate subset of types to be considered for a given program: this subset has to be large enough to correctly analyze the behaviour of the program and, at the same time, sufficiently small to be manipulated in realistic time. In previous work [4,18] for a call-by-name language, this subset is computed with the help of the control-flow analysis, which gives an over-approximation of the behaviour of the program. The naïve adaptation of this idea to a call-by-value language, however, does not work well. This is because the flow-information tends to be less accurate for call-by-value programs: in an application $t_1\,t_2$, one has to over-approximate the evaluation of both t_1 and t_2 in call-by-value, whereas in call-by-name t_2 itself is the accurate actual argument. We propose an algorithm (the *0-Control-Flow-Analysis (CFA) guided saturation algorithm*) that deeply integrates the type system and the 0-CFA. The integration reduces the inaccuracy of the flow analysis and makes the algorithm efficient, although it is technically more complicated.

We have implemented the algorithm, and confirmed through experiments that for large instances, our direct approach for model checking call-by-value Boolean programs outperforms the previous indirect approach of translating a call-by-value program to HORS and then model-checking the HORS.

The contributions of this paper are summarized as follows.

- A practical algorithm for the call-by-value reachability problem in direct style. The way to reduce type candidates using control-flow analysis is quite different from that of previous algorithms [4,18].
- The formalization of the algorithm and a proof of its correctness. The proof is more involved than the corresponding proof of the correctness of Tsukada and Kobayashi's algorithm [20] due to the flow-based optimization, and also than that of the correctness of the HorSat algorithm [4], due to the call-by-value evaluation strategy.
- Implementation and evaluation of the algorithm.

The rest of this paper is structured as follows. Section 2 defines the target language, its semantics, and the reachability problem. Section 3 gives an intersection type system that characterizes the reachability of a program. Section 4 describes the 0-CFA guided saturation algorithm, and proves its correctness. Section 5 describes the experimental results. Section 6 discusses related work, and the final section concludes the paper. Some proofs and figures are omitted in this version and are available in the long version [19].

2 Call-by-Value Reachability Problem

2.1 Target Language

We first introduce notations used in the rest of this paper. We write **Lab**, **Var**, and **Fun**, respectively for the countable sets of *labels*, *variables*, and *function symbols*. We assume that the meta-variable ℓ represents a label, the meta-variables x, y represent variables, and f, g represent function symbols. We write $\mathrm{dom}(g)$ for the domain set of a function g, and $\tilde{x}$ for a finite sequence like $x_1, \ldots, x_k$. Let ρ be a map. We denote $\rho[x \mapsto v]$ as the map that maps y to v if $x = y$ and that behaves as the same as ρ otherwise. We denote $\emptyset$ as both the empty set and the empty map, whose domain set is the empty set.

$$\begin{array}{rcl}
P \text{ (Programs)} & ::= & \mathbf{let\ rec}\ D : \mathcal{K}\ \mathbf{in}\ t \\
t \text{ (Terms)} & ::= & e^{\ell} \\
e \text{ (Expressions)} & ::= & b \mid p \mid x \mid f \mid \mathrm{op}(\tilde{t}) \mid \langle t_1, \ldots, t_k \rangle \mid \pi_i^k t \mid t_1\ t_2 \\
& \mid & t_1 \oplus t_2 \mid \mathbf{fail} \mid \Omega \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \mid \mathbf{assume}\ t_1; t_2 \\
p \text{ (Lambda-abstractions)} & ::= & \lambda x : \kappa.\ t \\
b \text{ (Booleans)} & ::= & \mathbf{true} \mid \mathbf{false} \\
\kappa \text{ (Sorts)} & ::= & \mathbf{bool} \mid \langle \kappa_1, \ldots, \kappa_k \rangle \mid \kappa_1 \to \kappa_2 \\
D \text{ (Global definitions)} & ::= & \{f_1 \mapsto p_1, \ldots, f_k \mapsto p_k\} \\
\mathcal{K} \text{ (Global sort environments)} & ::= & \{\ f_1 \mapsto \kappa_1, \ldots, f_k \mapsto \kappa_k\ \} \\
\Sigma \text{ (Local sort environments)} & ::= & \{\ x_1 \mapsto \kappa_1, \ldots, x_k \mapsto \kappa_k\ \}
\end{array}$$

Fig. 2. Syntax

The target language of the reachability analysis in this paper is a simply-typed, call-by-value lambda calculus with Booleans, tuples and global mutual recursions. The syntax of the language is given in Fig. 2. Each subterm is labeled with **Lab** in this language, for the control-flow analysis described later. We call *terms* for labeled ones, and *expressions* for unlabeled ones. The expression $\mathrm{op}(\tilde{t})$ is a Boolean operation, such as $t_1 \wedge t_2$, $t_1 \vee t_2$, and $\neg t$, and $\pi_i^k t$ is the i-th (zero-indexed) projection for the k-tuple t. The expression $t_1 \oplus t_2$ is a non-deterministic choice of t_1 and t_2. The terms Ω and **fail** represent divergence and failure, respectively. The assume-expression **assume** $t_1; t_2$ evaluates t_2 only if t_1 is evaluated to **true** (and diverges if t_1 is evaluated to **false**).

A *sort* is the simple type of a term, which is either Boolean sort **bool**, a tuple sort, or a function sort; we use the word "sort" to distinguish simple types from intersection types introduced later. A *local sort environment* and (resp. *global sort environment*) is a finite map from variables (resp. function symbols) to sorts. A *global definition* is a finite map from function symbols to lambda-expressions. A *program* consists of a global definition D, a global sort environment $\mathcal{K}$, and a term, called the *main term*.

Next, we define *well-sorted* terms. Let $\mathcal{K}$ be a global sort environment, Σ a local sort environment, and κ a sort. A *sort judgment* for a term t (resp. an expression e) is of the form $\mathcal{K}, \Sigma \vdash t : \kappa$ (resp. $\mathcal{K}, \Sigma \vdash e : \kappa$). The sort system of the target language is the standard simple type system with the following primitive types: **fail** $: \kappa$, $\Omega : \kappa$, **assume** $:$ **bool** $\to \kappa$, and $\oplus : \kappa \to \kappa \to \kappa$. The inference rules are given in the long version [19].

The *depth* of a sort κ, written $\mathtt{dep}(\kappa)$, is defined as follows: $\mathtt{dep}(\mathbf{bool}) = 1$, $\mathtt{dep}(\langle\kappa_1,\dots,\kappa_k\rangle) = \max(\mathtt{dep}(\kappa_1),\dots,\mathtt{dep}(\kappa_k))$, and $\mathtt{dep}(\kappa_1 \to \kappa_2) = 1 + \max(\mathtt{dep}(\kappa_1),\mathtt{dep}(\kappa_2))$. The *depth of a well-sorted term* t, written $\mathtt{dep}(t)$, is the maximum depth of sorts which appear in the derivation tree of $\mathcal{K}, \Sigma \vdash t : \kappa$. Let D be a global definition, and $\mathcal{K}$ be a global sort environment. We write $\vdash D : \mathcal{K}$ if $\mathrm{dom}(D) = \mathrm{dom}(\mathcal{K})$ and $\forall f \in \mathrm{dom}(D). \mathcal{K}, \emptyset \vdash D(f) : \mathcal{K}(f)$. We say program $P = \mathbf{let\ rec}\ D : \mathcal{K}\ \mathbf{in}\ t_0$ has sort κ if $\vdash D : \mathcal{K}$, and $\mathcal{K}, \emptyset \vdash t_0 : \kappa$. We say P is *well-sorted* if P has some sort κ. The *depth of a well-sorted program* P is the maximum depth of terms in P.

Example 1. Consider the program $P_1 = \mathbf{let\ rec}\ D_1 : \mathcal{K}_1\ \mathbf{in}\ t_1$ where:

$$\begin{aligned} D_1 &= \{\, f \mapsto \lambda(y : \mathbf{bool} \to \mathbf{bool}).\ t_f \,\}\ \ \mathcal{K}_1 = \{\, f \mapsto (\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool} \,\} \\ t_f &= (\mathbf{assume}\ (y^1\ \mathbf{true}^2)^3; (\mathbf{assume}\ (\neg(y^5\ \mathbf{true}^6)^7)^8; \mathbf{fail}^9)^{10})^{11} \\ t_1 &= (\mathbf{let}\ z = (\lambda(x : \mathbf{bool}).\ (\mathbf{true}^{12} \oplus \mathbf{false}^{13})^{14})^{15}\ \mathbf{in}\ (f^{16}\ z^{17})^{18})^{19} \end{aligned}$$

P_1 is well-sorted and has sort **bool**.

2.2 Semantics

We define the operational semantics of the language in the style of Nielson et al. [13]; this style of operational semantics is convenient for discussing flow analysis later. First, we define the following auxiliary syntactic objects:

$$\begin{aligned} e &::= \cdots \mid c \mid \mathbf{bind}\ \rho\ \mathbf{in}\ t \\ \rho\ \text{(Environments)} &::= \{\, x_1 \mapsto v_1, \dots, x_n \mapsto v_k \,\} \\ c\ \text{(Closures)} &::= \mathbf{close}\ p\ \mathbf{in}\ \rho \\ v\ \text{(Values)} &::= w^\ell \\ w\ \text{(Pre-values)} &::= b \mid f \mid c \mid \langle v_1, \dots, v_k \rangle \end{aligned}$$

The term **close** p **in** ρ represents a closure, and the term **bind** ρ **in** t evaluates t under the environment ρ. An *environment* is a finite map from variables to values. A value is either a Boolean, a function symbol, a closure, or a tuple of values. We note that values (resp. pre-values) are subclass of terms (resp. expressions). We extend the sort inference rules to support these terms as follows:

$$\frac{\mathcal{K} \vdash \rho : \Sigma' \qquad \mathcal{K}, \Sigma' \vdash p : \kappa}{\mathcal{K}, \Sigma \vdash \mathbf{close}\ p\ \mathbf{in}\ \rho : \kappa}\ (\textsc{Close}) \qquad \frac{\mathcal{K} \vdash \rho : \Sigma' \qquad \mathcal{K}, \Sigma' \vdash t : \kappa}{\mathcal{K}, \Sigma \vdash \mathbf{bind}\ \rho\ \mathbf{in}\ t : \kappa}\ (\textsc{Bind})$$

$$\frac{\mathrm{dom}(\rho) = \mathrm{dom}(\Sigma) \qquad \forall x \in \mathrm{dom}(\rho).\ \mathcal{K}, \emptyset \vdash \rho(x) : \Sigma(x)}{\mathcal{K} \vdash \rho : \Sigma}\ (\textsc{Env})$$

A sort judgment for environments is of the form $\mathcal{K} \vdash \rho : \Sigma$, which means that for each binding $x \mapsto v$ in ρ, v has type $\Sigma(x)$.

Next, we define reduction relations. We fix some well-sorted program $P = \mathbf{let\ rec}\ D : \mathcal{K}\ \mathbf{in}\ e_0$. Let ρ be an environment, and Σ be a local sort environment such that $\mathcal{K} \vdash \rho : \Sigma$. The reduction relation for terms is of the form $\rho \vdash_D t \longrightarrow t'$, where $\mathcal{K}, \Sigma \vdash t : \kappa$ for some sort κ. The reduction rules are given in Fig. 3. In rule (Op-2), $[\![\mathrm{op}]\!]$ denotes the Boolean function that corresponds to each operation op, and $FV(p)$ denotes the set of free variables of p. We write $\rho \vdash_D t \longrightarrow^* t'$ for the reflexive and transitive closure of $\rho \vdash_D t_1 \longrightarrow t_2$.

$$\frac{\rho(x) = w^{\ell_0}}{\rho \vdash_D x^\ell \longrightarrow w^\ell}\ (\textsc{Var}) \qquad \frac{\rho \vdash_D t \longrightarrow t'}{\rho \vdash_D \mathrm{op}(\tilde{v}, t, \tilde{t})^\ell \longrightarrow \mathrm{op}(\tilde{v}, t', \tilde{t})^\ell}\ (\textsc{Op-1}) \qquad \frac{b' = [\![\mathrm{op}]\!](\tilde{b})}{\rho \vdash_D \mathrm{op}(\tilde{b})^\ell \longrightarrow b'^\ell}\ (\textsc{Op-2})$$

$$\frac{\rho \vdash_D t \longrightarrow t'}{\rho \vdash_D \langle \tilde{v}, t, \tilde{t} \rangle^\ell \longrightarrow \langle \tilde{v}, t', \tilde{t} \rangle^\ell}\ (\textsc{Tuple}) \qquad \frac{\rho \vdash_D t \longrightarrow t'}{\rho \vdash_D (\pi_i^k t)^\ell \longrightarrow (\pi_i^k t')^\ell}\ (\textsc{Proj-1})$$

$$\frac{v = \langle w_0^{\ell_0}, \dots, w_{k-1}^{\ell_{k-1}} \rangle^{\ell'}}{\rho \vdash_D (\pi_i^k v)^\ell \longrightarrow w_i^\ell}\ (\textsc{Proj-2}) \qquad \frac{\rho' = \{\, x \mapsto \rho(x) \mid x \in \mathrm{FV}(p) \,\}}{\rho \vdash_D p^\ell \longrightarrow (\mathbf{close}\ p\ \mathbf{in}\ \rho')^\ell}\ (\textsc{Fun})$$

$$\frac{\rho \vdash_D t_1 \longrightarrow t_1'}{\rho \vdash_D (t_1\ t_2)^\ell \longrightarrow (t_1'\ t_2)^\ell}\ (\textsc{App-1}) \qquad \frac{c = \mathbf{close}\ \lambda x : \kappa.\ t\ \mathbf{in}\ \rho'}{\rho \vdash_D (c^{\ell_1}\ v_2)^\ell \longrightarrow (\mathbf{bind}\ \rho'[x \mapsto v_2]\ \mathbf{in}\ t)^\ell}\ (\textsc{App-3})$$

$$\frac{\rho \vdash_D t_2 \longrightarrow t_2'}{\rho \vdash_D (v_1\ t_2)^\ell \longrightarrow (v_1\ t_2')^\ell}\ (\textsc{App-2}) \qquad \frac{(\lambda x : \kappa.\ t) = D(f)}{\rho \vdash_D (f^{\ell_1}\ v_2)^\ell \longrightarrow (\mathbf{bind}\ [x \mapsto v_2]\ \mathbf{in}\ t)^\ell}\ (\textsc{App-4})$$

$$\frac{\rho \vdash_D t_1 \longrightarrow t_1'}{\rho \vdash_D (\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2)^\ell \longrightarrow (\mathbf{let}\ x = t_1'\ \mathbf{in}\ t_2)^\ell}\ (\textsc{Let-1})$$

$$\frac{}{\rho \vdash_D (\mathbf{let}\ x = v_1\ \mathbf{in}\ t_2)^\ell \longrightarrow (\mathbf{bind}\ \rho[x \mapsto v_1]\ \mathbf{in}\ t_2)^\ell}\ (\textsc{Let-2})$$

$$\frac{i \in \{\, 1, 2 \,\}}{\rho \vdash_D (e_1^{\ell_1} \oplus e_2^{\ell_2})^\ell \longrightarrow e_i^\ell}\ (\textsc{Br}) \qquad \frac{\rho \vdash_D t_1 \longrightarrow t_1'}{\rho \vdash_D (\mathbf{assume}\ t_1; t_2)^\ell \longrightarrow (\mathbf{assume}\ t_1'; t_2)^\ell}\ (\textsc{Assume-1})$$

$$\frac{}{\rho \vdash_D (\mathbf{assume}\ \mathbf{true}^{\ell_1}; e_2^{\ell_2})^\ell \longrightarrow e_2^\ell}\ (\textsc{Assume-2})$$

$$\frac{\rho' \vdash_D t \longrightarrow t'}{\rho \vdash_D (\mathbf{bind}\ \rho'\ \mathbf{in}\ t)^\ell \longrightarrow (\mathbf{bind}\ \rho'\ \mathbf{in}\ t')^\ell}\ (\textsc{Bind-1}) \qquad \frac{}{\rho \vdash_D (\mathbf{bind}\ \rho'\ \mathbf{in}\ w^{\ell_1})^\ell \longrightarrow w^\ell}\ (\textsc{Bind-2})$$

Fig. 3. Reduction relation

2.3 Reachability Problem

We are interested in the reachability problem: whether a program P may execute the command **fail** We define the set of *error expressions*, called **Err**, as follows:[1]

[1] Note that the terms like **assume false**; t and Ω are not error expressions. They are intended to model divergent terms, although they are treated as stuck terms in the operational semantics for a technical convenience.

$$\begin{aligned}\phi \text{ (Error expr.)} ::= {} & \mathbf{fail} \mid \mathbf{let}\ x = \phi^\ell\ \mathbf{in}\ t_2 \mid \mathbf{bind}\ \rho\ \mathbf{in}\ \phi^\ell \mid \langle \tilde{v}, \phi^\ell, \tilde{t}\rangle \mid \mathrm{op}(\tilde{v}, \phi^\ell, \tilde{t}) \\ \mid {} & \mathbf{assume}\ \phi^\ell; t \mid \phi^\ell\ t \mid v\ \phi^\ell.\end{aligned}$$

Then, the reachability problem is defined as follows.

Definition 1 (Reachability Problem). *A program $P = \mathbf{let\ rec}\ D : \mathcal{K}\ \mathbf{in}\ t_0$ is* unsafe *if $\emptyset \vdash_D t_0 \longrightarrow^* \phi^\ell$ holds for some $\phi \in \mathbf{Err}$. A well-sorted program P is called* safe *if P is not unsafe. Given a well-sorted program, the task of the* reachability problem *is to decide whether the program is safe.*

Example 2 For example, $P_1 = \mathbf{let\ rec}\ D_1 : \mathcal{K}_1\ \mathbf{in}\ t_1$ in Example 1 is unsafe, and the program $P_2 = \mathbf{let\ rec}\ D_1 : \mathcal{K}_1\ \mathbf{in}\ t_2$ below is safe.

$$t_2 = (\mathbf{let}\ w = (\mathbf{true}^{20} \oplus \mathbf{false}^{21})^{22}\ \mathbf{in}\ (f^{23}\ (\lambda(x : \mathbf{bool}).\ w^{24})^{25})^{26})^{27}$$

3 Intersection Type System

In this section, we present an intersection type system that characterizes the unsafety of programs, which is an extension of Tsukada and Kobayashi's type system [20].

The sets of *value types* σ and *term types* τ are defined by:

$$\sigma ::= \mathbf{true} \mid \mathbf{false} \mid \langle \sigma_1, \ldots, \sigma_k \rangle \mid \bigwedge_{i \in I} (\sigma_i \to \tau_i) \qquad \tau ::= \sigma \mid \mathbf{fail}$$

Value types are those for values, and term types are for terms, as the names suggest. Intuitively the type **true** describes the value **true**. The type of the form $\langle \sigma_1, \ldots, \sigma_k \rangle$ describes a tuple whose i-th element has type σ_i. A type of the form $\bigwedge_{i \in I} (\sigma_i \to \tau_i)$ represents a function that returns a term of type τ_i if the argument has type σ_i for each $i \in I$. Here, we suppose that I be some finite set. We write $\bigwedge \emptyset$ if I is the empty set. A term type is either a value type or the special type **fail**, which represents a term that is evaluated to an error term. We also call a *local type environment* Δ (resp. *a global type environment* Γ) for a finite map from variables (resp. function symbols) to value types.

The *refinement relations* $\sigma :: \kappa$ and $\tau :: \kappa$ for value/term types are defined by the following rules:

$$\frac{}{b :: \mathbf{bool}} \qquad \frac{\sigma_i :: \kappa_1 \qquad \tau_i :: \kappa_2 \qquad \text{for each } i}{(\bigwedge_i \sigma_i \to \tau_i) :: (\kappa_1 \to \kappa_2)}$$

$$\frac{\sigma_i :: \kappa_i \qquad \text{for each } i}{\langle \sigma_1, \ldots, \sigma_k \rangle :: \langle \kappa_1, \ldots, \kappa_k \rangle} \qquad \frac{}{\mathbf{fail} :: \kappa} \quad .$$

We naturally extend this refinement relation to those for local/global type environments and denote $\Delta :: \Sigma$ and $\Gamma :: \mathcal{K}$.

$$\frac{\Gamma, \Delta \vdash e : \tau}{\Gamma, \Delta \vdash e^{\ell} : \tau}\ (\textsc{Term}) \qquad \frac{\Gamma, \Delta \vdash t_i : \sigma_i \text{ for each } 1 \leq i \leq k}{\Gamma, \Delta \vdash t_1 \dots t_k : \sigma_1 \dots \sigma_k}\ (\textsc{Seq})$$

$$\frac{\Gamma, \Delta \vdash t_1 \dots t_{l-1} : \tilde{\sigma} \qquad \Gamma, \Delta \vdash t_l : \mathbf{fail} \qquad \text{for some } 0 \leq l \leq k}{\Gamma, \Delta \vdash t_1 \dots t_k : \mathbf{fail}}\ (\textsc{Seq-F})$$

$$\frac{}{\Gamma, \Delta \vdash x : \Delta(x)}\ (\textsc{Var}) \qquad \frac{}{\Gamma, \Delta \vdash b : b}\ (\textsc{Bool}) \qquad \frac{\Gamma, \Delta \vdash \tilde{t} : \tilde{b}}{\Gamma, \Delta \vdash \mathrm{op}(\tilde{t}) : [\![\mathrm{op}]\!](\tilde{b})}\ (\textsc{Op})$$

$$\frac{\Gamma, \Delta \vdash \tilde{t} : \tilde{\sigma}}{\Gamma, \Delta \vdash \langle \tilde{t} \rangle : \langle \tilde{\sigma} \rangle}\ (\textsc{Tuple}) \qquad \frac{\Gamma, \Delta \vdash t : \langle \sigma_0, \dots, \sigma_{k-1} \rangle}{\Gamma, \Delta \vdash \pi_i^k t : \sigma_i}\ (\textsc{Proj})$$

$$\frac{\Gamma, \Delta \vdash \tilde{t} : \mathbf{fail}}{\Gamma, \Delta \vdash \mathrm{op}(\tilde{t}) : \mathbf{fail}}\ (\textsc{Op-F}) \qquad \frac{\Gamma, \Delta \vdash \tilde{t} : \mathbf{fail}}{\Gamma, \Delta \vdash \langle \tilde{t} \rangle : \mathbf{fail}}\ (\textsc{Tuple-F}) \qquad \frac{\Gamma, \Delta \vdash t : \mathbf{fail}}{\Gamma, \Delta \vdash \pi_i^k t : \mathbf{fail}}\ (\textsc{Proj-F})$$

$$\frac{\Gamma, \Delta[x \mapsto \sigma_i] \vdash t : \tau_i \qquad \sigma_i :: \kappa \qquad \text{for each } i \in I}{\Gamma, \Delta \vdash \lambda x : \kappa.\ t : \bigwedge_{i \in I}(\sigma_i \to \tau_i)}\ (\textsc{Fun}) \qquad \frac{\Gamma, \Delta \vdash t_1, t_2 : \mathbf{fail}}{\Gamma, \Delta \vdash t_1\ t_2 : \mathbf{fail}}\ (\textsc{App-F})$$

$$\frac{\Gamma, \Delta \vdash t_1 : \tau}{\Gamma, \Delta \vdash t_1 \oplus t_2 : \tau}\ (\textsc{Br-1}) \qquad \frac{\Gamma, \Delta \vdash t_2 : \tau}{\Gamma, \Delta \vdash t_1 \oplus t_2 : \tau}\ (\textsc{Br-2}) \qquad \frac{}{\Gamma, \Delta \vdash \mathbf{fail} : \mathbf{fail}}\ (\textsc{Fail})$$

$$\frac{\Gamma, \Delta \vdash t_1 : \bigwedge_{i \in I}(\sigma_i \to \tau_i) \qquad \Gamma, \Delta \vdash t_2 : \sigma_j \text{ for some } j \in I}{\Gamma, \Delta \vdash t_1\ t_2 : \tau_j}\ (\textsc{App})$$

$$\frac{\Gamma, \Delta \vdash t_1 : \sigma_1 \qquad \Gamma, \Delta[x \mapsto \sigma_1] \vdash t_2 : \tau}{\Gamma, \Delta \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : \tau}\ (\textsc{Let}) \qquad \frac{\Gamma, \Delta \vdash t_1 : \mathbf{fail}}{\Gamma, \Delta \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : \mathbf{fail}}\ (\textsc{Let-F})$$

$$\frac{\Gamma, \Delta \vdash t_1 : \mathbf{true} \qquad \Delta \vdash t_2 : \tau}{\Gamma, \Delta \vdash \mathbf{assume}\ t_1; t_2 : \tau}\ (\textsc{Assume}) \qquad \frac{\Gamma, \Delta \vdash t_1 : \mathbf{fail}}{\Gamma, \Delta \vdash \mathbf{assume}\ t_1; t_2 : \mathbf{fail}}\ (\textsc{Assume-F})$$

$$\frac{\Gamma \vdash \rho : \Delta' \qquad \Gamma, \Delta' \vdash p : \sigma}{\Gamma, \Delta \vdash \mathbf{close}\ p\ \mathbf{in}\ \rho : \sigma}\ (\textsc{Close}) \qquad \frac{\Gamma \vdash \rho : \Delta' \qquad \Gamma, \Delta' \vdash t : \tau}{\Delta \vdash \mathbf{bind}\ \rho\ \mathbf{in}\ t : \tau}\ (\textsc{Bind})$$

$$\frac{\mathrm{dom}(\Delta) = \mathrm{dom}(\rho) \qquad \Gamma \vdash \rho(x) : \Delta(x) \text{ for each } x \in \mathrm{dom}(\Delta)}{\Gamma \vdash \rho : \Delta}\ (\textsc{Env})$$

Fig. 4. Typing rules

There are four kinds of type judgments in the intersection type system;

- $\Gamma, \Delta \vdash t : \tau$ for term t;
- $\Gamma, \Delta \vdash e : \tau$ for expression e;
- $\Gamma, \Delta \vdash \tilde{t} : \tilde{\sigma}$ or $\Gamma, \Delta \vdash \tilde{t} : \mathbf{fail}$ for sequence $\tilde{t}$; and
- $\Gamma \vdash \rho : \Delta$ for environment ρ.

The typing rules for those judgments are given in Fig. 4. Intuitively, the type judgment for terms represents "under-approximation" of the evaluation of the term. The judgment $\Gamma, \Delta \vdash t : \sigma$ intuitively means that there is a reduction $\rho \vdash_D t \longrightarrow^* v$ for some value v of type σ, and $\Gamma, \Delta \vdash t : \mathbf{fail}$ means that $\rho \vdash_D t \longrightarrow^* \phi^{\ell}$ for some error expression ϕ. For example, for the term $t_1 = \langle \mathbf{true} \oplus \mathbf{false}, \mathbf{true} \rangle^{\ell}$, the judgments $\Gamma, \Delta \vdash t_1 : \langle \mathbf{true}, \mathbf{true} \rangle$ and $\Gamma, \Delta \vdash t_1 : \langle \mathbf{false}, \mathbf{true} \rangle$ should hold because there are reductions $\rho \vdash_D t_1 \longrightarrow^* \langle \mathbf{true}, \mathbf{true} \rangle^{\ell}$ and $\rho \vdash_D t_1 \longrightarrow^* \langle \mathbf{false}, \mathbf{true} \rangle^{\ell}$. Furthermore, for the term $t_2 = (\textbf{let } x = \textbf{true} \oplus \textbf{false in assume } x; \textbf{fail})^\ell$, $\mathit{\Gamma}, \mathit{\Delta} \vdash t_2 : \textbf{fail}$ because $\rho \vdash_D t_2 \longrightarrow^* (\textbf{bind } \rho[x \mapsto \textbf{true}] \textbf{ in fail})^\ell$. We remark that a term that always diverges (e.g. $\mathit{\Omega}$ and $\textbf{assume false}; t$) does not have any types. The judgments $\mathit{\Gamma}, \mathit{\Delta} \vdash \tilde{t} : \tilde{\sigma}$ and $\mathit{\Gamma}, \mathit{\Delta} \vdash \tilde{t} : \textbf{fail}$ are auxiliary judgments, which correspond to the evaluation strategy that evaluates $\tilde{t}$ from left to right. For example, the rule (Seq-F) means that the evaluation $\tilde{t} = t_1 \ldots t_k$ fails (e.g. $\mathit{\Gamma}, \mathit{\Delta} \vdash \tilde{t} : \textbf{fail}$) if and only if some t_i fails (e.g. $\mathit{\Gamma}, \mathit{\Delta} \vdash t_i : \textbf{fail}$), and $t_0, \ldots, t_{i-1}$ are evaluated to some values $\tilde{v}$ (e.g. $\mathit{\Gamma}, \mathit{\Delta} \vdash t_1, \ldots, t_{i-1} : \tilde{\sigma}$). The judgment for environments $\mathit{\Gamma}, \mathit{\Delta} \vdash \rho : \mathit{\Delta}$ represents that for each binding $[x \mapsto v]$ in ρ, v has type $\mathit{\Delta}(x)$.

The type system above is an extension of Tsukada and Kobayashi's one [20]. The main differences are:

- Our target language supports tuples as first-class values, while tuples may occur only as function arguments in their language. By supporting them, we avoid hyper-exponential explosion of the number of types caused by the CPS-transformation to eliminate first-class tuples.
- Our target language also supports let-expressions. Although it is possible to define them as syntactic sugar, supporting them as primitives makes our type inference algorithm more efficient.

We define some operators used in the rest of this section. Let σ and σ' be value types that are refinements of some function sort. The intersection of σ and σ', written as $\sigma \wedge \sigma'$, is defined by:

$$\bigwedge_{i \in I}(\sigma_i \to \tau_i) \wedge \bigwedge_{j \in J}(\sigma_j \to \tau_j) = \bigwedge_{k \in (I \cup J)}(\sigma_k \to \tau_k),$$

where $\sigma = \bigwedge_{i \in I}(\sigma_i \to \tau_i)$ and $\sigma' = \bigwedge_{j \in J}(\sigma_j \to \tau_j)$. Let D be a global definition, and $\mathit{\Gamma}$ and $\mathit{\Gamma}'$ be global type environments. We say $\mathit{\Gamma}'$ is a D-expansion of $\mathit{\Gamma}$, written as $\mathit{\Gamma} \lhd_D \mathit{\Gamma}'$, if the following condition holds:

$$\mathit{\Gamma} \lhd_D \mathit{\Gamma}' \iff \begin{array}{l} \mathrm{dom}(\mathit{\Gamma}) = \mathrm{dom}(\mathit{\Gamma}'), \\ \forall f \in \mathrm{dom}(\mathit{\Gamma}). \exists \sigma.\ \mathit{\Gamma}, \emptyset \vdash D(f) : \sigma \text{ and } \mathit{\Gamma}'(f) = (\mathit{\Gamma}(f) \wedge \sigma) \end{array}$$

This expansion soundly computes types of each recursive function. Intuitively, $\mathit{\Gamma} \lhd_D \mathit{\Gamma}'$ means that, assuming $\mathit{\Gamma}$ is a sound type environment for D, $\mathit{\Gamma}'(f)$ is a sound type of f because $\mathit{\Gamma}'(f)$ is obtained from $\mathit{\Gamma}(f)$ by adding a valid type of $D(f)$. We write $\mathit{\Gamma}_D^\top$ for the environment $\{f \mapsto \bigwedge \emptyset \mid f \in \mathrm{dom}(D)\}$, which corresponds to approximating D as $D^\top = \{f \mapsto \lambda x : \kappa.\ \mathit{\Omega} \mid f \in \mathrm{dom}(D)\}$. It is always safe to approximate the behaviour of D with $\mathit{\Gamma}_D^\top$. We write $\lhd_D^*$ for the reflexive and transitive closure of $\lhd_D$. We say $\mathit{\Gamma}$ is *sound for* D if $\mathit{\Gamma}_D^\top \lhd_D^* \mathit{\Gamma}$.

Theorem 1 indicates that the intersection type system characterizes the reachability of Boolean programs. The proof is similar to the proof of the corresponding theorem for Tsukada and Kobayashi's type system [20]: see the long version [19].

Theorem 1. *Let* $P = \textbf{let rec } D : \mathcal{K} \textbf{ in } t_0$ *be a well-sorted program.* P *is unsafe if and only if there is a global type environment* $\mathit{\Gamma}$ *that is sound for D, and that* $\mathit{\Gamma}, \emptyset \vdash t_0 : \textbf{fail}$.

According to this theorem, the reachability checking problem is solved by saturation-based algorithms. For example, it is easily shown that the following naïve saturation function $\mathcal{F}_D$ is sufficient for deciding the reachability.

$$\mathcal{F}_D(\Gamma)(f) = \Gamma(f) \wedge \bigwedge \left\{ \sigma \to \tau \;\middle|\; \begin{array}{l} D(f) = \lambda x : \kappa.\ t, \sigma :: \kappa, \\ \Gamma, [x \mapsto \sigma] \vdash t : \tau \end{array} \right\}$$

The saturation function is effectively computable. To compute the second operand of $\wedge$ in the definition of $\mathcal{F}_D(\Gamma)(f)$, it suffices to pick each σ such that $\sigma :: \kappa$, and computes τ such that $\Gamma, [x \mapsto \sigma] \vdash t : \tau$. Note that there are only finitely many σ such that $\sigma :: \kappa$. Given a well-sorted program **let rec** $D : \mathcal{K}$ **in** t_0, let $\Gamma_0 = \Gamma_D^\top$ and $\Gamma_{i+1} = \mathcal{F}_D(\Gamma_i)$. The sequence $\Gamma_0, \Gamma_1, \ldots, \Gamma_m, \ldots$ converges for some m, because $\Gamma_i \lhd_D \Gamma_{i+1}$ for each i, and $\lhd_D$ is a partial order on the (finite) set of type environments. Thus, the reachability is decided by checking whether $\Gamma_m, \emptyset \vdash t_0 : \mathbf{fail}$ holds.

4 The 0-CFA Guided Saturation Algorithm

In the following discussion, we fix some well-sorted program $P =$ **let rec** D : $\mathcal{K}$ **in** t_0. We assume that all variables bound in lambda-expressions or let-expressions in P are distinct from each other, and that all the labels in P are also distinct from each other. Therefore, we assume each variable x has the corresponding sort, and we write $\mathrm{sort}(x)$ for it.

This section presents an efficient algorithm for deciding the reachability problem, based on the type system in the previous section. Unfortunately, the naïve algorithm presented in Sect. 3 is impractical, mainly due to the (Fun) rule:

$$\frac{\Gamma, \Delta[x \mapsto \sigma_i] \vdash t : \tau_i \qquad \sigma_i :: \kappa \qquad \text{for each } i \in I}{\Gamma, \Delta \vdash \lambda x : \kappa.\ t : \bigwedge_{i \in I}(\sigma_i \to \tau_i)} \quad (\textsc{Fun})$$

The rule tells us how to enumerate the type judgments for $\lambda x : \kappa.\ t$ from those for t, but there are a huge number of candidate types of the argument x because they are only restricted to have a certain sort κ; when the depth of κ is k, the number of candidate types is k-fold exponential. Therefore, we modify the type system to reduce irrelevant type candidates.

4.1 The $\hat{\delta}$-Guided Type System

A *flow type environment* is a function that maps a variable x to a set of value types that are refinement of $\mathrm{sort}(x)$. Let Γ be a global type environment, $\hat{\delta}$ be a flow type environment, and Δ be a local type environment. We define the $\hat{\delta}$-guided type judgment of the form either $\Gamma, \Delta \vdash_{\hat{\delta}} t : \tau$ or $\Gamma, \Delta \vdash_{\hat{\delta}} e : \tau$. The typing rules for this judgment are the same as that of $\Gamma, \Delta \vdash t : \tau$, except for (Fun), which is replaced by the following rule:

$$\frac{S = \left\{ (\sigma, \tau) \;\middle|\; \sigma \in \hat{\delta}(x), \Gamma, \Delta[x \mapsto \sigma] \vdash_{\hat{\delta}} t : \tau \right\}}{\Gamma, \Delta \vdash_{\hat{\delta}} \lambda x : \kappa.\ t : \bigwedge_{(\sigma,\tau)\in S}(\sigma \to \tau)} \qquad \text{(FUN')}$$

This modified rule derives the "strongest" type of the lambda-abstraction, assuming $\hat{\delta}(x)$ is an over-approximation of the set of types bound to x. This type system, named *the $\hat{\delta}$-guided type system*, is built so that the type judgments are deterministic for values, lambda-abstractions and environments.

Proposition 1. *Suppose $\Gamma :: \mathcal{K}$. Then,*

- $\mathcal{K}, \emptyset \vdash v : \kappa$ *implies* $\exists! \sigma. \sigma :: \kappa \wedge \Gamma, \emptyset \vdash_{\hat{\delta}} v : \sigma$,
- $\mathcal{K}, \emptyset \vdash p : \kappa$ *implies* $\exists! \sigma. \sigma :: \kappa \wedge \Gamma, \emptyset \vdash_{\hat{\delta}} p : \sigma$, *and*
- $\mathcal{K} \vdash \rho : \Sigma$ *implies* $\exists! \Delta. \Delta :: \Sigma \wedge \Gamma \vdash_{\hat{\delta}} \rho : \Delta$.

Thereby, we write $[\![v]\!]_{\Gamma,\hat{\delta}}$, $[\![p]\!]_{\Gamma,\hat{\delta}}$ and $[\![\rho]\!]_{\Gamma,\hat{\delta}}$ for the value type of value v, lambda-abstraction p, and environment ρ, respectively.

We define the $\hat{\delta}$-guided saturation function $\mathcal{G}_D(\hat{\delta}, \Gamma)$ as follows:

$$\mathcal{G}_D(\hat{\delta}, \Gamma)(f) = \Gamma(f) \wedge [\![D(f)]\!]_{\Gamma,\hat{\delta}}$$

It is easily shown that the soundness theorem of $\hat{\delta}$-guided type system holds.

Theorem 2 (Soundness). *Let $P =$ **let rec** $D : \mathcal{K}$ **in** t_0 be a well-sorted program. Let $\hat{\delta}_0, \hat{\delta}_1, \ldots$ be a sequence of flow type environments. We define a sequence of global type environments $\Gamma_0, \Gamma_1, \ldots$ as follows: (i) $\Gamma_0 = \Gamma_D^{\top}$, and (ii) $\Gamma_{i+1} = \mathcal{G}_D(\hat{\delta}_i, \Gamma_i)$ for each $i \geq 0$. The program P is unsafe if there is some m such that $\Gamma_m, \emptyset \vdash_{\hat{\delta}_m} t_0 :$* **fail**.

However, the completeness of the $\hat{\delta}$-guided type system depends on the flow environments used during saturation. For example, if we use the largest flow type environment, that is, $\hat{\delta}(x) = \{ \sigma \mid \sigma :: \mathrm{sort}(x) \}$, we have the completeness, but we lose the efficiency. We have to find a method to compute a sufficiently large flow type environment $\hat{\delta}$ such that the $\hat{\delta}$-guided type system achieves both the completeness and the efficiency.

In the call-by-name case, a sufficient condition on $\hat{\delta}$ to guarantee the completeness can be formalized in terms of flow information [4]. For each function call $t_1\ t_2$, we just need to require that $\hat{\delta}(x) \supseteq \{ \sigma \mid \Gamma, \Delta \vdash_{\hat{\delta}} t_2 : \sigma \}$ for each possible value $\lambda x.\ t$ of t_1.

However, in the call-by-value case, the condition on $\hat{\delta}$ is more subtle because the actual value bound to argument x is not t_2 itself but an evaluation result of t_2. In order to prove that the $\hat{\delta}$-guided type system is complete, it is required that $\hat{\delta}(x)$ contains all the types of the values bound to x during the evaluation,[2] i.e. $\hat{\delta}(x) \supseteq \{ [\![v]\!]_{\Gamma,\hat{\delta}} \mid \rho \vdash_D t_2 \longrightarrow^* v \}$. Therefore, we have to prove that

[2] In the call-by-name case, this property immediately follows from the condition $\hat{\delta}(x) \supseteq \{ \sigma \mid \Gamma, \Delta \vdash_{\hat{\delta}} t_2 : \sigma \}$ because t_2 is not evaluated before the function call.

$\{ [\![v]\!]_{\Gamma,\hat{\delta}} \mid \rho \vdash_D t_2 \longrightarrow^* v \} \supseteq \{ \sigma \mid \Gamma, \Delta \vdash_{\hat{\delta}} t_2 : \sigma \}$, but this fact follows from the completeness of the $\hat{\delta}$-guided type system, which causes a circular reasoning.

In the rest of this section, we first formalize 0-CFA for our target language, propose our 0-CFA guided saturation algorithm, and prove the correctness of the algorithm.

4.2 0-CFA

We adopt the formalization of 0-CFA by Nielson et al. [13].

An *abstract value* is defined by:

$$av \text{ (abstract values) } ::= \texttt{bool} \mid p \mid f \mid \langle av_1, \dots, av_k \rangle.$$

The set of abstract values is denoted as $\widehat{\mathbf{Value}}$. An abstract value is regarded as a value without environments. The abstract value of a value v, written $\hat{v}$, is defined by:

$$\widehat{w^\ell} = \hat{w} \qquad \hat{b} = \texttt{bool} \qquad \hat{f} = f$$
$$\widehat{\mathbf{close}\ p\ \mathbf{in}\ \rho} = p \qquad \widehat{\langle v_1, \dots, v_k \rangle} = \langle \widehat{v_1}, \dots, \widehat{v_k} \rangle.$$

An *abstract cache* is a function from **Lab** to $\mathcal{P}(\widehat{\mathbf{Value}})$, and an *abstract environment* is a function from **Var** to $\mathcal{P}(\widehat{\mathbf{Value}})$. Let $\hat{C}$ be an abstract cache, and $\hat{\rho}$ be an abstract environment. We define the relations $(\hat{C}, \hat{\rho}) \models_D e^\ell$ and $(\hat{C}, \hat{\rho}) \models_D \rho$, which represents $(\hat{C}, \hat{\rho})$ is an *acceptable* CFA result of the term e^ℓ and the environment ρ, respectively.

The relations are co-inductively defined by the rules given in Fig. 5. In the (TUPLE) rule, $\hat{C}(\ell_1) \otimes \cdots \otimes \hat{C}(\ell_k)$ means the set $\{ \langle \hat{v}_1, \dots, \hat{v}_k \rangle \mid \forall i. \hat{v}_i \in \hat{C}(\ell_i) \}$. In the (PROJ) rule, $\pi_i^k(\hat{C}(\ell_1)) = \{ \hat{v}_i \mid \langle \hat{v}_0, \dots, \hat{v}_{k-1} \rangle \in \hat{C}(\ell_1) \}$. The relation $(\hat{C}, \hat{\rho}) \models_D e^\ell$ is defined so that if e^ℓ is evaluated to a value v, then the abstract value of v is in $\hat{C}(\ell)$. The relation $(\hat{C}, \hat{\rho}) \models_D \rho$ means that for each binding $x \mapsto v$ in ρ, $\hat{\rho}(x)$ contains the abstract value of v.

4.3 The 0-CFA Guided Saturation Algorithm

We propose a method to compute a sufficiently large $\hat{\delta}$ so that the $\hat{\delta}$-guided type system would be complete. Let $\hat{C}$ be an abstract cache. We define two relations $(\hat{C}, \hat{\delta}) \models_{D,\Gamma} (t, \Delta)$, and $(\hat{C}, \hat{\delta}) \models_{D,\Gamma} \rho$. The relation $(\hat{C}, \hat{\delta}) \models_\Gamma (t, \Delta)$ means intuitively that, during any evaluations of t under an environment ρ such that $\Gamma \vdash \rho : \Delta$, the type of values bound to variable x is approximated by $\hat{\delta}(x)$. The derivation rules for those relations are given in Fig. 6. We regard these rules as an algorithm to saturate $\hat{\delta}$, given $\hat{C}$, Δ and t. The algorithm basically traverses the term t with possible Δ using $\hat{\delta}$-guided type system as dataflow, and propagates types to $\hat{\delta}$ using the rule (APP): if t is an function call $e_1^\ell\ t_2$, the algorithm enumerates each lambda abstraction $\lambda x : \kappa.\ t_0$ that e_1^ℓ may be evaluated to by using $\hat{C}$, and propagates each type σ of t_2 (i.e. $\Gamma, \Delta \vdash_{\hat{\delta}} t : \sigma$) to $\hat{\delta}(x)$.

$$\frac{\hat{C}(\ell) \ni \mathtt{bool}}{(\hat{C},\hat{\rho}) \models_D b^\ell}\ (\textsc{Bool}) \qquad \frac{\hat{C}(\ell) \supseteq \hat{\rho}(x)}{(\hat{C},\hat{\rho}) \models_D x^\ell}\ (\textsc{Var}) \qquad \frac{}{(\hat{C},\hat{\rho}) \models_D \mathbf{fail}^\ell}\ (\textsc{Fail}) \qquad \frac{}{(\hat{C},\hat{\rho}) \models_D \Omega^\ell}\ (\textsc{Omega})$$

$$\frac{\hat{C}(\ell) \ni f \quad \begin{array}{c} D(f) = \lambda x : \kappa.\ t \\ (\hat{C},\hat{\rho}) \models_D t \end{array}}{(\hat{C},\hat{\rho}) \models_D f^\ell}\ (\textsc{Tfun}) \qquad \frac{\begin{array}{c}(\hat{C},\hat{\rho}) \models_D e^{\ell_1} \\ \hat{C}(\ell) \supseteq \pi_i^k(\hat{C}(\ell_1))\end{array}}{(\hat{C},\hat{\rho}) \models_D (\pi_i^k e^{\ell_1})^\ell}\ (\textsc{Proj})$$

$$\frac{(\hat{C},\hat{\rho}) \models_D \rho \qquad (\hat{C},\hat{\rho}) \models_D p}{(\hat{C},\hat{\rho}) \models_D (\mathbf{close}\ p\ \mathbf{in}\ \rho)^\ell}\ (\textsc{Close}) \qquad \frac{\hat{C}(\ell) \ni (\lambda x : \kappa.\ t) \qquad (\hat{C},\hat{\rho}) \models_D t}{(\hat{C},\hat{\rho}) \models_D (\lambda x : \kappa.\ t)^\ell}\ (\textsc{Fun})$$

$$\frac{(\hat{C},\hat{\rho}) \models_D t_i \text{ for each } i \qquad \hat{C}(\ell) \ni \mathtt{bool}}{(\hat{C},\hat{\rho}) \models_D \mathrm{op}(t_1,\ldots,t_k)^\ell}\ (\textsc{Op}) \qquad \frac{\begin{array}{c}(\hat{C},\hat{\rho}) \models_D e_i^{\ell_i} \text{ for each } i \\ \hat{C}(\ell) \supseteq \hat{C}(\ell_1) \otimes \cdots \otimes \hat{C}(\ell_k)\end{array}}{(\hat{C},\hat{\rho}) \models_D \langle e_1^{\ell_1},\ldots,e_k^{\ell_k}\rangle^\ell}\ (\textsc{Tuple})$$

$$\frac{\begin{array}{ll}(\hat{C},\hat{\rho}) \models_D e_1^{\ell_1} & \forall(\lambda x : \kappa.\ e^{\ell_0}) \in (\hat{C}(\ell_1) \cup \{\, D(f) \mid f \in \hat{C}(\ell_1)\,\}). \\ (\hat{C},\hat{\rho}) \models_D e_2^{\ell_2} & \rho(x) \supseteq \hat{C}(\ell_2) \wedge \hat{C}(\ell) \supseteq \hat{C}(\ell_0)\end{array}}{(\hat{C},\hat{\rho}) \models_D (e_1^{\ell_1}\ e_2^{\ell_2})^\ell}\ (\textsc{App})$$

$$\frac{\begin{array}{c}(\hat{C},\hat{\rho}) \models_D e_1^{\ell_1} \\ (\hat{C},\hat{\rho}) \models_D \rho\end{array} \quad \hat{C}(\ell) \supseteq \hat{C}(\ell_1)}{(\hat{C},\hat{\rho}) \models_D (\mathbf{bind}\ \rho\ \mathbf{in}\ e_1^{\ell_1})^\ell}\ (\textsc{Bind}) \qquad \frac{\begin{array}{ll}(\hat{C},\hat{\rho}) \models_D e_1^{\ell_1} & \hat{C}(\ell) \supseteq \hat{C}(\ell_1) \\ (\hat{C},\hat{\rho}) \models_D e_2^{\ell_2} & \hat{C}(\ell) \supseteq \hat{C}(\ell_2)\end{array}}{(\hat{C},\hat{\rho}) \models_D (e_1^{\ell_1} \oplus e_2^{\ell_2})^\ell}\ (\textsc{Br})$$

$$\frac{\begin{array}{c}(\hat{C},\hat{\rho}) \models_D e_1^{\ell_1}\ (\hat{C},\hat{\rho}) \models_D e_2^{\ell_2} \\ \hat{\rho}(x) \supseteq \hat{C}(\ell_1)\ \ \hat{C}(\ell) \supseteq \hat{C}(\ell_2)\end{array}}{(\hat{C},\hat{\rho}) \models_D \mathbf{let}\ x = e_1^{\ell_1}\ \mathbf{in}\ e_2^{\ell_2}}\ (\textsc{Let}) \qquad \frac{\begin{array}{c}(\hat{C},\hat{\rho}) \models_D t_1 \\ (\hat{C},\hat{\rho}) \models_D e_2^{\ell_2}\end{array} \quad \hat{C}(\ell) \supseteq \hat{C}(\ell_2)}{(\hat{C},\hat{\rho}) \models_D (\mathbf{assume}\ t_1; e_2^{\ell_2})^\ell}\ (\textsc{Assume})$$

$$\frac{(\hat{C},\hat{\rho}) \models_D w^\ell \qquad \hat{\rho}(x) \supseteq \hat{C}(\ell) \qquad \text{for each binding } x \mapsto w^\ell \text{ in } \rho}{(\hat{C},\hat{\rho}) \models_D \rho}\ (\textsc{Env})$$

Fig. 5. 0-CFA rules

Algorithm 1 shows our algorithm for the reachability problem, named *the 0-CFA guided saturation algorithm*. Given a well-sorted program $P = \mathbf{let\ rec}\ D : \mathcal{K}\ \mathbf{in}\ t_0$, the algorithm initializes Γ_0 with $\Gamma_D^\top$, computes a 0-CFA result $(\hat{C},\hat{\rho})$ such that $(\hat{C},\hat{\rho}) \models_D t$, sets $i = 0$, and enters the main loop. In the main loop, it computes $\hat{\delta}_i$ such that $(\hat{C},\hat{\delta}_i) \models_{D,\Gamma_i} t_0$, and then, sets Γ_{i+1} with $\mathcal{G}_D(\hat{\delta}_i,\Gamma_i)$ and increments i. The algorithm outputs "UNSAFE" if $\Gamma_i \vdash_{\hat{\delta}_i} t_0 : \mathbf{fail}$ holds for some i. Otherwise, the main loop eventually breaks when $\Gamma_i = \Gamma_{i-1}$ holds, and then, the algorithm outputs "SAFE".

We explain how the saturation algorithm runs for the program P_1 in Example 1. Let ℓ_1 and ℓ_2 be the labels of the first application of y and the second application of y in function f. A result of 0-CFA would be $\hat{C}(\ell_1) = \hat{C}(\ell_2) = \lambda(x : \mathbf{bool}).\ \mathbf{true} \oplus \mathbf{false}$. Let $\Gamma_0 = \{\, f \mapsto \bigwedge \emptyset \,\}$. Then, $\hat{\delta}_0$ would be

$$\hat{\delta}_0(y) = \{\, \bigwedge \emptyset, (\mathbf{true} \to \mathbf{true}) \wedge (\mathbf{true} \to \mathbf{false}) \,\} \qquad \hat{\delta}_0(x) = \{\, \mathbf{true} \,\}.$$

Therefore, $\Gamma_0, \emptyset \vdash_{\hat{\delta}_0} D_1(f) : (\mathbf{true} \to \mathbf{true}) \wedge (\mathbf{true} \to \mathbf{false}) \to \mathbf{fail}$ holds, and it would be $\Gamma_1 = \{\, f : (\mathbf{true} \to \mathbf{true}) \wedge (\mathbf{true} \to \mathbf{false}) \to \mathbf{fail} \,\}$. In the next

$$\frac{}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (b^{\ell},\Delta)}\ (\textsc{Bool}) \qquad \frac{}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (x^{\ell},\Delta)}\ (\textsc{Var}) \qquad \frac{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t,\Delta)}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} ((\pi_i^k t)^{\ell},\Delta)}\ (\textsc{Proj})$$

$$\frac{}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (\mathbf{fail}^{\ell},\Delta)}\ (\textsc{Fail}) \qquad \frac{D(f) = \lambda x:\kappa.\ t \quad (\hat{C},\hat{\delta}) \models_{D,\Gamma} (t,[x \mapsto \sigma]) \text{ for each } \sigma \in \hat{\delta}(x)}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (f^{\ell},\Delta)}\ (\textsc{Tfun})$$

$$\frac{}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (\Omega^{\ell},\Delta)}\ (\textsc{Omega}) \qquad \frac{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t,\Delta[x \mapsto \sigma]) \text{ for each } \sigma \in \hat{\delta}(x)}{(\hat{C},\hat{\rho}) \models_{D,\Gamma} ((\lambda x:\kappa.\ t)^{\ell},\Delta)}\ (\textsc{Fun})$$

$$\frac{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_i,\Delta) \text{ for each } i}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (\mathrm{op}(t_1,\dots,t_k)^{\ell},\Delta)}\ (\textsc{Op}) \qquad \frac{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_i,\Delta) \text{ for each } i}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (\langle t_1,\dots,t_k\rangle^{\ell},\Delta)}\ (\textsc{Tuple})$$

$$\frac{\begin{array}{ll}(\hat{C},\hat{\delta}) \models_{D,\Gamma} (e_1^{\ell_1},\Delta), & \forall(\lambda x:\kappa.\ t) \in (\hat{C}(\ell_1) \cup \{\, D(f) \mid f \in \hat{C}(\ell_1)\,\}) \\ (\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_2,\Delta) & \hat{\delta}(x) \supseteq \{\, \sigma \mid \Gamma,\Delta \vdash_{\hat{\delta}} t_2 : \sigma \,\}\end{array}}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} ((e_1^{\ell_1}\ t_2)^{\ell},\Delta)}\ (\textsc{App})$$

$$\frac{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_1,\Delta) \qquad (\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_2,\Delta[x \mapsto \sigma]) \text{ for each } \Gamma,\Delta \vdash_{\hat{\delta}} t_1 : \sigma}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} ((\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2)^{\ell},\Delta)}\ (\textsc{Let})$$

$$\frac{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_1,\Delta) \qquad \Gamma,\Delta \vdash_{\hat{\delta}} t_1 : \mathbf{true} \implies (\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_2,\Delta)}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} ((\mathbf{assume}\ t_1; t_2)^{\ell},\Delta)}\ (\textsc{Assume})$$

$$\frac{(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t,[\![\rho]\!]_{\Gamma,\hat{\delta}}) \qquad (\hat{C},\hat{\delta}) \models_{D,\Gamma} \rho}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} ((\mathbf{bind}\ \rho\ \mathbf{in}\ t)^{\ell},\Delta)}\ (\textsc{Bind}) \qquad \frac{(\hat{C},\hat{\delta}) \models_{D,\Gamma} \rho \qquad (\hat{C},\hat{\delta}) \models_{\Gamma} (p^{\ell},[\![\rho]\!]_{\Gamma,\hat{\delta}})}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} ((\mathbf{close}\ p\ \mathbf{in}\ \rho)^{\ell},\Delta)}\ (\textsc{Close})$$

$$\frac{\begin{array}{c}(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_1,\Delta) \\ (\hat{C},\hat{\delta}) \models_{D,\Gamma} (t_2,\Delta)\end{array}}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} ((t_1 \oplus t_2)^{\ell},\Delta)}\ (\textsc{Br}) \qquad \frac{\forall x \in \mathrm{dom}(\rho).\ (\hat{C},\hat{\delta}) \models_{D,\Gamma} (\rho(x),\emptyset)}{(\hat{C},\hat{\delta}) \models_{D,\Gamma} \rho}\ (\textsc{Env})$$

Fig. 6. Derivation rules for $(\hat{C},\hat{\delta}) \models_{D,\Gamma} (t,\Delta)$ and $(\hat{C},\hat{\delta}) \models_{D,\Gamma} \rho$

Algorithm 1. The 0-CFA guided saturation algorithm

```
function IsSafe($P = \mathbf{let\ rec}\ D : \mathcal{K}\ \mathbf{in}\ t_0$)
    $\Gamma_0 := \Gamma_D^{\top}$
    Compute $(\hat{C},\hat{\rho})$ such that $(\hat{C},\hat{\rho}) \models_D t_0$
    $i := 0$
    repeat
        Compute $\hat{\delta}_i$ such that $(\hat{C},\hat{\delta}_i) \models_{D,\Gamma_i} (t_0,\emptyset)$
        $\Gamma_{i+1} = \mathcal{G}_D(\hat{\delta}_i,\Gamma_i)$
        $i := i + 1$
        if $\Gamma_{i-1},\emptyset \vdash_{\hat{\delta}_{i-1}} t_0 : \mathbf{fail}$ then
            return Unsafe
        end if
    until $\Gamma_{i-1} = \Gamma_i$
    return Safe
end function
```

iteration, there are no updates, i.e. $\hat{\delta}_1 = \hat{\delta}_0$ and $\Gamma_1 = \Gamma_0$. Because $\Gamma_1, \emptyset \vdash_{\hat{\delta}_1} t_1 :$ **fail** holds, the algorithm outputs "Unsafe".

4.4 Correctness of the 0-CFA Guided Saturation Algorithm

We prove the correctness of Algorithm 1. If the algorithm outputs "Unsafe", the given program is unsafe by using Theorem 2. In order to justify the case that the algorithm outputs "Safe", we prove the completeness of the $\hat{\delta}$-guided type system.

First, the following lemma indicates that $(\hat{C}, \hat{\rho}) \models_D t$ and $(\hat{C}, \hat{\delta}) \models_{D,\Gamma} (t, \Delta)$ satisfy subject reduction, and also that the $\hat{\delta}$-guided type system satisfies subject expansion. This lemma solves the problem of circular reasoning discussed at the end of Sect. 4.1.

Lemma 1. *Let Γ be a global type environment such that $\Gamma = \mathcal{G}_D(\hat{\delta}, \Gamma)$. Suppose that $(\hat{C}, \hat{\rho}) \models_D t_1$, $(\hat{C}, \hat{\rho}) \models_D \rho$, $\rho \vdash_D t_1 \longrightarrow t_2$, $(\hat{C}, \hat{\delta}) \models_{D,\Gamma} \rho$, and $(\hat{C}, \hat{\delta}) \models_{D,\Gamma} (t_1, \Delta)$, where $\Delta = [\![\rho]\!]_{\Gamma,\hat{\delta}}$. Then, (i) $(\hat{C}, \hat{\delta}) \models_D t_2$, (ii) $(\hat{C}, \hat{\delta}) \models_{D,\Gamma} (t_2, \Delta)$, and (iii) for any term type τ, $\Gamma, \Delta \vdash_{\hat{\delta}} t_2 : \tau$ implies $\Gamma, \Delta \vdash_{\hat{\delta}} t_1 : \tau$.*

We use the fact that $\hat{\delta}$-guided type system derives **fail** for error terms.

Lemma 2. *Let ϕ be a well-sorted error expression. Then, $\Gamma, \emptyset \vdash_{\hat{\delta}} \phi :$ **fail**.*

Then, we have the following completeness theorem, which justifies the correctness of Algorithm 1.

Theorem 3. *Let $P =$ **let rec** $D : \mathcal{K}$ **in** t_0 be a well-sorted program, Γ be a global type environment such that $\Gamma :: \mathcal{K}$ and $\Gamma = \mathcal{G}_D(\hat{\delta}, \Gamma)$. Suppose that $(\hat{C}, \hat{\rho}) \models_D t_0$, and $(\hat{C}, \hat{\delta}) \models_{D,\Gamma} (t_0, \emptyset)$. If $\Gamma, \emptyset \not\vdash_{\hat{\delta}} t_0 :$ **fail** then P is safe.*

Proof. We prove the contraposition. Assume that P is unsafe, i.e., that there is a sequence $e_0 \ldots e_n$ such that $e_0^\ell = t_0$, $\emptyset \vdash_D e_i^\ell \longrightarrow e_{i+1}^\ell$ for each $0 \leq i \leq n-1$, and that e_n^ℓ is an error term. We have $\forall \tau. \Gamma, \emptyset \vdash_{\hat{\delta}} e_n^\ell : \tau \implies \Gamma, \emptyset \vdash_{\hat{\delta}} t_0 : \tau$ by induction on n and using Lemma 1. By Lemma 2, $\Gamma, \emptyset \vdash_{\hat{\delta}} e_n^\ell :$ **fail**. Therefore, we have $\Gamma, \emptyset \vdash_{\hat{\delta}} t_0 :$ **fail**. □

5 Implementation and Experiments

5.1 Benchmarks and Environment

We have implemented a reachability checker named HiBoch for call-by-value Boolean programs. In order to evaluate the performance of our algorithm, we prepared two benchmarks. The first benchmark consists of Boolean programs generated by a CEGAR-based verification system for ML programs. More precisely, we prepared fourteen instances of verification problems for ML programs, which have been manually converted from the MoCHi benchmark [17], and passed them to our prototype CEGAR-based verification system, which uses

HiBoch as a backend reachability checker. During each CEGAR cycle, the system generates an instance of the reachability problem for Boolean programs by predicate abstraction, and we used these problem instances for the first benchmark.

The second benchmark consists of a series of Boolean programs generated by a template named "Flow", which was manually designed to clarify the differences between the direct and indirect styles. More details on this benchmark are given in the long version [19].

We compared our direct method with the previous indirect method, which converts Boolean programs to HORS and checks the reachability with a higher-order model checker. We use HorSat [4] as the main higher-order model checker in the indirect method; since HorSat also uses a 0-CFA-based saturation algorithm (but for HORS, not for Boolean programs), we believe that HorSat is the most appropriate target of comparison for evaluating the difference between the direct/indirect approaches. We also report the results of the indirect method using the other state-of-the-art higher-order model checkers HorSat2 [10] and Preface [16], but one should note that the difference of the performance may not properly reflect that between the direct/indirect approaches, because HorSat2 uses a different flow analysis and Preface is not based on saturation.

The experimental environment was as follows. The machine spec is 2.3 GHz Intel Core i7 CPU, 16 GB RAM. Our implementation was compiled with the Glasgow Haskell Compiler, version 7.10.3, HorSat and HorSat2 were compiled with the OCaml native-code compiler, version 4.02.1, and Preface was run on Mono JIT compiler version 3.2.4. The running times of each model checker were limited to 200 s.

5.2 Experimental Result

Figures 7 and 8 show the experimental results. The horizontal axis is the size of Boolean programs, measured on the size of the abstract syntax trees, and the vertical axis is the elapsed time of each model checker, excluding the elapsed times for converting the reachability problem instances to the higher-order model checking instances.

For the first benchmark, HiBoch solves all the test cases in a few seconds. For the instances of size within 5000, HorSat2 is the fastest, and HiBoch is the second fastest, which is 4–7 times faster than HorSat (and also Preface). For the instances of size over 5000, HiBoch is the fastest[3] by an order of magnitude. We regard the reason of this result as the fact that these instances have larger arity (where the arity means the number of function arguments). The indirect style approach suffers from huge numbers of combinations between argument types. Our direct approach reduces many irrelevant combinations using the structure of call-by-value programs, which is lost during the CPS-transformation.

For the second benchmark, as we expected, HiBoch clearly outperforms the indirect approaches, even the one using HorSat2.

[3] Unfortunately, we could not measure the elapsed time of HorSat2 for some large instances because it raised stack-overflow exceptions.

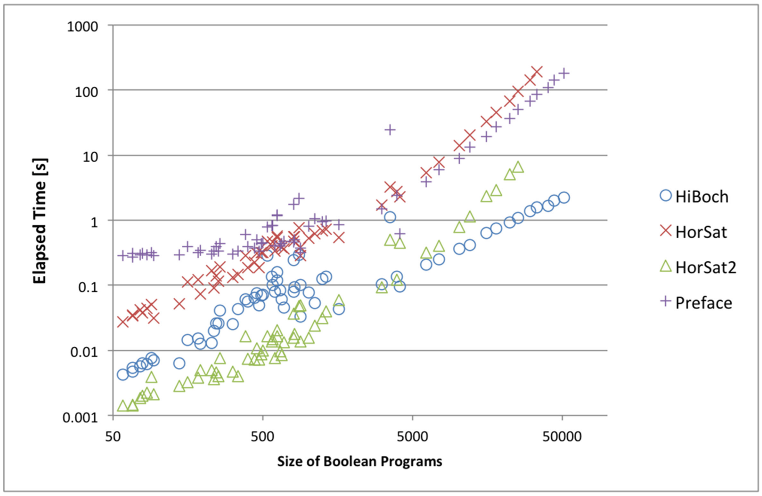

Fig. 7. Experimental result for MoCHi benchmark

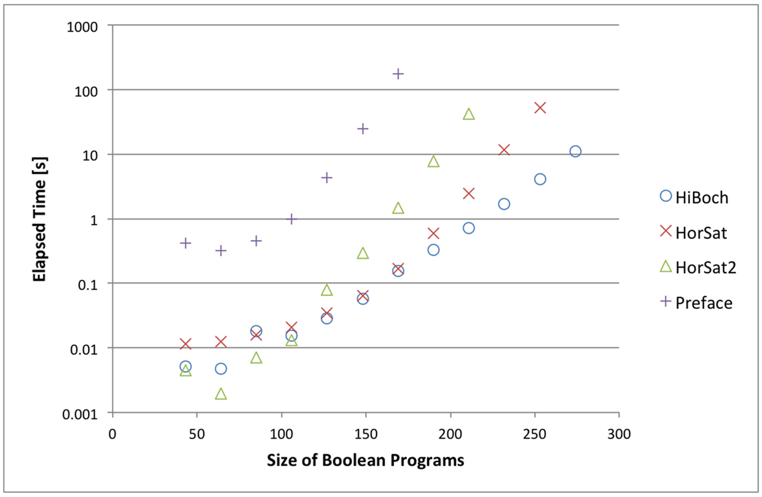

Fig. 8. Experimental result for flow benchmark

6 Related Work

As mentioned already, the reachability of higher-order call-by-value Boolean programs has been analyzed by a combination of CPS-transformation and higher-order model checking [11,17]. Because the naïve CPS-transformation algorithm often generates too complex HORS, Sato et al. [17] proposed a method called *selected CPS transformation*, in which insertion of some redundant continuations is avoided. The experiments reported in Sect. 5 adapt this selective

CPS transformation, but the indirect method still suffers from the complexity due to the CPS transformation.

Tsukada and Kobayashi [20] studied the complexity of the reachability problem, and showed that the problem is k-EXPTIME complete for *depth*-k programs. They also introduced an intersection type system and a type inference algorithm, which are the basis of our work. However, their algorithm has been designed just for proving an upper-bound of the complexity; the algorithm is impractical in the sense that it always suffers from the k-EXPTIME bottleneck, while our 0-CFA guided algorithm does not.

For *first-order* Boolean programs, Ball and Rajamani [2] proposed a path-sensitive, dataflow algorithm and implemented `Bebop` tool, which is used as a backend of SLAM [1]. It is not clear whether and how their algorithm can be extended to deal with *higher-order* Boolean programs.

Flow-based optimizations have been used in recent model checking algorithms for higher-order recursion schemes [3,4,16,18]. However, naïve application of such optimizations to call-by-value language would be less accurate because we need to estimate the evaluation result of not only functions but also their arguments. Our method employs the intersection type system to precisely represent the evaluation results.

Some of the recent higher-order model checkers [10,16] use more accurate flow information. For example, Preface [16] dynamically refines flow information using type-based abstraction. We believe it is possible to integrate more accurate flow analysis [5–7] also into our algorithm.

7 Conclusion

We have proposed a direct algorithm for the reachability problem of higher-order Boolean programs, and proved its correctness. We have confirmed through experiments that our direct approach improves the performance of the reachability analysis.

We are now developing a direct-style version of MoCHi, a fully automated software model checker for OCaml programs, on top of our reachability checker for Boolean programs, and plan to compare the overall performance with the indirect style. We expect that avoiding CPS transformations also benefits the predicate discovery phase.

Acknowledgment. This work was supported by JSPS KAKENHI Grant Numbers JP16J01038 and JP15H05706.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: technology transfer of formal methods inside microsoft. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24756-2_1

2. Ball, T., Rajamani, S.K.: Bebop: a path-sensitive interprocedural dataflow engine. In: Proceedings of PASTE 2001, pp. 97–103. ACM (2001)
3. Broadbent, C.H., Carayol, A., Hague, M., Serre, O.: C-SHORe: acollapsible approach to higher-order verification. In: Proceedings of ICFP 2013, pp. 13–24 (2013)
4. Broadbent, C.H., Kobayashi, N.: Saturation-based model checking of higher-order recursion schemes. In: Proceedings of CSL 2013, LIPIcs, vol. 23, pp. 129–148 (2013)
5. Gilray, T., Lyde, S., Adams, M.D., Might, M., Horn, D.V.: Pushdown control-flow analysis for free. In: Proceedings of POPL 2016, pp. 691–704. ACM (2016)
6. Horn, D.V., Might, M.: Abstracting abstract machines. In: Proceedings of ICFP 2010, pp. 51–62. ACM (2010)
7. Johnson, J.I., Horn, D.V.: Abstracting abstract control. In: Proceedings of DLS 2014, pp. 11–22. ACM (2014)
8. Kobayashi, N.: Model-checking higher-order functions. In: Proceedings of PPDP 2009, pp. 25–36. ACM (2009)
9. Kobayashi, N.: Model checking higher-order programs. J. ACM **60**(3) (2013)
10. Kobayashi, N.: HorSat2: a saturation-based higher-order model checker. A toolpaper under submission (2015). http://www-kb.is.s.u-tokyo.ac.jp/~koba/horsat2
11. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of PLDI 2011, pp. 222–233. ACM (2011)
12. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 392–411. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54833-8_21
13. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, New York (1999)
14. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: Proceedings of LICS 2006, pp. 81–90. IEEE Computer Society Press (2006)
15. Ong, C.H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Proceedings of POPL 2011, pp. 587–598. ACM (2011)
16. Ramsay, S.J., Neatherway, R.P., Ong, C.L.: A type-directed abstraction refinement approach to higher-order model checking. In: Proceedings of POPL 2014, pp. 61–72. ACM (2014)
17. Sato, R., Unno, H., Kobayashi, N.: Towards a scalable software model checker for higher-order programs. In: Proceedings of PEPM 2013, pp. 53–62. ACM (2013)
18. Terao, T., Kobayashi, N.: A ZDD-based efficient higher-order model checking algorithm. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 354–371. Springer, Heidelberg (2014). doi:10.1007/978-3-319-12736-1_19
19. Terao, T., Tsukada, T., Kobayashi, N.: Higher-order model checking in direct style (2016). http://www-kb.is.s.u-tokyo.ac.jp/~terao/papers/aplas16.pdf
20. Tsukada, T., Kobayashi, N.: Complexity of model-checking call-by-value programs. In: Muscholl, A. (ed.) FoSSaCS 2014. LNCS, vol. 8412, pp. 180–194. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54830-7_12