A ZDD-based Efficient Higher-order Model Checking Algorithm

Taku Terao and Naoki Kobayashi

The University of Tokyo

Abstract. The model checking of higher-order recursion schemes, aka. higher-order model checking, has recently been applied to automated verification of higher-order programs. Despite its extremely high worst-case complexity, practical algorithms have been developed that work well for typical inputs that arise in program verification. Even the state-of-the-art algorithms are, however, not scalable enough for verification of thousands or millions of lines of programs. We, therefore, propose a new higher-order model checking algorithm. It is based on Broadbent and Kobayashi's type and saturation-based algorithm HorSat, but we make two significant modifications. First, unlike HorSat, we collect flow information (which is necessary for optimization) in linear time by using a sub-transitive flow graph. Thanks to this, the resulting algorithm runs in almost linear time under a fixed-parameter assumption. Secondly, we employ zero-suppressed binary decision diagrams to efficiently represent and propagate type information. We have confirmed through experiments that the new algorithm is more scalable for several families of inputs than the state-of-the-art higher-order model checkers HorSat and Preface.

1 Introduction

Higher-order model checking is the problem of deciding whether the (possibly infinite) tree generated by a given higher-order recursion scheme (HORS) satisfies a given property [19]. Higher-order model checking has recently been applied to automatic verification of higher-order functional program [9, 13, 20, 12, 14].

A major challenge in applying higher-order model checking to practice is to develop an *efficient* higher-order model checker. Actually, the higher-order model checking problem is k-EXPTIME complete for order-k HORS [19, 11], so there is no hope to obtain an algorithm that works well for all the inputs. Nevertheless, several practical algorithms have been developed and implemented, which run reasonably fast for many typical inputs [9, 8, 18, 3, 21]. The state-of-the-art higher-order model checkers HorSat [3] and Preface [21] can handle HORS consisting of hundreds of lines of rewriting rules (and it has been reported [21] that Preface works even for thousands of lines of HORS for a specific problem instance). Despite the recent advance, they are still not scalable enough to be applied to verification of thousands or millions of lines of programs.

In the present paper, we improve the HorSat algorithm [3] in two significant ways. HorSat computes (a finite representation of) the backward closure of error configurations (i.e., the set of terms that generate error trees) by using intersection types, and checks whether the initial configuration belongs to the set. It has two main bottlenecks: one is the flow analysis (based on 0CFA [23]) employed to compute only a relevant part of the backward closure. In theory, (the known upper-bound of) the worst-case complexity of the flow analysis is almost cubic time [16], whereas the other part of the HorSat algorithm is actually fixed-parameter linear time.[1] The other bottleneck is that the number of intersection types used for representing a set of terms may blow up quickly. This blow up immediately slows down the whole algorithm, since each saturation step (for computing the backward closure by iteratively computing the backward image of a set of terms) picks and processes each type one by one. To overcome the first problem, we employ a linear-time *sub-transitive* control flow analysis (which constructs a graph whose transitive closure is a flow graph) [5] and use it for the optimization. This guarantees that the whole algorithm runs in time linear in the size of HORS, under the same fixed-parameter assumption as before. To address the second problem, we represent a set of intersection types using a zero-suppressed binary decision diagram (ZDD) [17], and develop a new saturation algorithm that can process a set of intersection types (represented in the form of ZDD) simultaneously.

We have implemented the new algorithm mentioned above and confirmed that it scales better (with respect to the size of HORS) than HorSat and Preface for several classes of inputs parametrized by the size of HORS.

The rest of the paper is structured as follows. Section 2 reviews the higher-order model checking problem, the model checking algorithm HorSat, and ZDD. Section 3 describes our new algorithm. Section 4 reports experiments. Section 5 discusses related work and Section 6 concludes the paper.

2 Preliminaries

We review higher-order recursion schemes (HORS) and higher-order model checking [19]. To save the definitions, we consider here a specialized version of higher-order model checking called co-trivial ATA model checking of HORS [3].

2.1 Higher-order Recursion Schemes and Co-Trivial ATA Model Checking

The set of **sorts**, written **Sorts**, is defined by: $\kappa ::= \mathsf{o} \mid \kappa_1 \to \kappa_2$. Intuitively, o describes trees, and $\kappa_1 \to \kappa_2$ describes functions from κ_1 to κ_2. A **sorted alphabet** is a map from a finite set of symbols to **Sorts**. The *arity* and *order* of **Sorts** are defined by:

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

[1] Actually, in the previously reported implementation of HorSat [3], the other part also took more than linear time due to the naive implementation. In the present work, we have also improved on that point.

Let X be a sorted alphabet. The (family of) sets $\mathbf{Terms}_{X,\kappa}$ of *applicative terms* of sort κ over X is inductively defined by: (i) If $X(a) = \kappa$, then $a \in \mathbf{Terms}_{X,\kappa}$; and (ii) If $t_1 \in \mathbf{Terms}_{X,\kappa_2 \to \kappa}$ and $t_2 \in \mathbf{Terms}_{X,\kappa_2}$, then $t_1 t_2 \in \mathbf{Terms}_{X,\kappa}$. We write $\mathbf{Terms}_X$ for the union of $\mathbf{Terms}_{X,\kappa}$ for all sorts.

Definition 1 (Higher-order recursion schemes (HORS)). *A* ***higher-order recursion scheme*** *is a tuple $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where: (i) Σ and $\mathcal{N}$ are sorted alphabets, where $\mathcal{N}(S) = o$, $\mathrm{dom}(\Sigma) \cap \mathrm{dom}(\mathcal{N}) = \emptyset$, and $\mathsf{order}(\Sigma(a)) \leq 1$ for every $a \in \mathrm{dom}(\Sigma)$; (ii) $\mathcal{R}$ is a set of rewriting rules of the form $F\,x_1\,\cdots\,x_n \to t$ where $\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_n \to o$ and $t \in \mathbf{Terms}_{\Sigma \cup \mathcal{N} \cup \{\,x_1:\kappa_1,\ldots,x_n:\kappa_n\,\},o}$. We require that $\mathcal{R}$ has exactly one rule for each $F \in \mathrm{dom}(\mathcal{N})$. The* ***reduction relation*** *$t_1 \longrightarrow_{\mathcal{G}} t_2$ is the least binary relation on $\mathbf{Terms}_{\Sigma\cup\mathcal{N},o}$ that satisfies: (i) $F\ t_1 \cdots t_n \longrightarrow_{\mathcal{G}} [t_1/x_1,\ldots,t_n/x_n]t$ if $F\,x_1\,\cdots\,x_n \to t \in \mathcal{R}$, and (ii) $a\ t_1 \cdots t_i \cdots t_n \longrightarrow_{\mathcal{G}} a\ t_1 \cdots t_i' \cdots t_n$ if $t_i \longrightarrow_{\mathcal{G}} t_i'$ and $a \in \mathrm{dom}(\Sigma)$. The* ***value tree*** *of $\mathcal{G}$, written $\mathbf{Tree}(\mathcal{G})$, is the least upper bound of $\{\,t^{\perp} \mid S \longrightarrow_{\mathcal{G}}^{*} t\,\}$ (with respect to the least precongruence $\sqsubseteq$ that satisfies $\perp \sqsubseteq t$ for every tree t), where $t^{\perp}$ is defined by $(F\ t_1 \cdots t_n)^{\perp} = \perp$ for each $F \in \mathrm{dom}(\mathcal{N})$ and $(a\ t_1 \cdots t_n)^{\perp} = a\ t_1^{\perp} \cdots t_n^{\perp}$ for each $a \in \mathrm{dom}(\Sigma)$. We call each x_i in $F\,x_1\,\cdots\,x_n \to t$ a* ***variable.*** *We assume that all variables are distinct from each other. $\mathcal{X}$ denotes the sorted alphabet of all variables.*

Intuitively, each symbol $a \in \mathrm{dom}(\Sigma)$ (called a terminal symbol) is a tree constructor of arity $\mathsf{arity}(\Sigma(a))$, and $F \in \mathrm{dom}(\mathcal{R})$ (called a non-terminal symbol) is a (higher-order) function on trees defined by the rewriting rules.

Example 1. Consider the HORS $\mathcal{G}_0 = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where $\Sigma = \{\mathsf{a} : \mathsf{o} \to \mathsf{o} \to \mathsf{o}, \mathsf{b} : \mathsf{o} \to \mathsf{o}, \mathsf{c} : \mathsf{o}\}$, $\mathcal{N} = \{S : \mathsf{o}, F : (\mathsf{o} \to \mathsf{o}) \to \mathsf{o}, T : (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}\}$, and $\mathcal{R}$ consists of the rules:

$$S \to F\,(T\,\mathsf{b}) \qquad F\,f \to \mathsf{a}\,(f\,\mathsf{c})\,(F\,(T\,f)) \qquad T\,g\,x \to g(g(x))$$

S is reduced as follows.

$$S \longrightarrow F\,(T\,\mathsf{b}) \longrightarrow \mathsf{a}\,(T\,\mathsf{b}\,\mathsf{c})\,(F\,(T\,(T\,\mathsf{b}))) \longrightarrow \mathsf{a}\,(\mathsf{b}(\mathsf{b}\,\mathsf{c}))\,(F\,(T\,(T\,\mathsf{b}))) \longrightarrow \cdots$$

It generates an infinite tree having a path $\mathsf{a}^k\mathsf{b}^{2^k}\mathsf{c}$ for every $k \geq 1$.

Higher-order model checking is the problem of deciding whether $\mathbf{Tree}(\mathcal{G})$ satisfies a given tree property. We use alternating tree automata (for finite trees) to describe the tree property. We consider below an element of $\mathbf{Terms}_{\Sigma,\mathsf{o}}$ as a tree.

Definition 2 (Alternating Tree Automata (ATA)). *An* ***alternating tree automaton*** *is a tuple (Σ, Q, δ, q_I) where: (i) Σ is a sorted alphabet; (ii) Q is a finite set; (iii) $\delta \subseteq Q \times \mathrm{dom}(\Sigma) \times 2^{\mathbb{N}\times Q}$ such that whenever $(q, a, U) \in \delta$ and $(i, q') \in U$, $1 \leq i \leq \mathsf{arity}(\Sigma(a))$; and (iv) $q_I \in Q$. A* ***configuration*** *is a set of pairs of the form $(t, q) \in \mathbf{Terms}_{\Sigma,o} \times Q$, and the* ***transition relation*** *over configurations is defined by:*

$$C \cup \{\,(a\,t_1\,\cdots\,t_k, q)\,\} \longrightarrow C \cup \{\,(t_i, q') \mid (i, q') \in U\,\}\ \ \textit{(if } (q, a, U) \in \delta\textit{).}$$

*A tree $t \in \mathbf{Terms}_{\Sigma,o}$ is **accepted** if $\{ (q_I, t) \} \longrightarrow^* \emptyset$. We write $\mathcal{L}(\mathcal{A})$ for the set of trees accepted by $\mathcal{A}$. For an ATA $\mathcal{A} = (\Sigma, Q, \delta, q_I)$ (with $\perp \notin \mathrm{dom}(\Sigma)$), we write $\mathcal{A}^\perp$ for $(\Sigma \cup \{ \perp \mapsto \mathtt{o} \}, Q, \delta, q_I)$.*

Example 2. Consider the automaton $\mathcal{A}_0 = (\Sigma, \{ q_0, q_1 \}, \delta, q_0)$ where Σ is the same as that of Example 1, and δ is:

$$\begin{array}{l} \{ (q_i, \mathtt{a}, \{ (j, q_i) \}) \mid j \in \{ 1, 2 \}, i \in \{ 0, 1 \} \} \\ \cup \{ (q_0, \mathtt{b}, \{ (1, q_1) \}), (q_1, \mathtt{b}, \{ (1, q_0) \}), (q_1, \mathtt{c}, \emptyset) \} \end{array}$$

It accepts all the trees that have a finite path containing an odd number of b's.

We can now define a special case of higher-order model checking called the co-trivial model checking of HORS [3].

Definition 3 (Co-trivial Model Checking for HORS). *We write $\mathcal{G} \models \mathcal{A}$ if there exists a term t such that $S \longrightarrow^* t$ and $t^\perp \in \mathcal{L}(\mathcal{A}^\perp)$. The co-trivial ATA model checking of HORS is the problem of deciding whether $\mathcal{G} \models \mathcal{A}$ holds, given an ATA $\mathcal{A} = (\Sigma, Q, \delta, q_I)$ and a HORS $\mathcal{G}$ as input.*

Intuitively, the ATA describes the property of *invalid* trees, and the condition "$S \longrightarrow^* t$ and $t^\perp \in \mathcal{L}(\mathcal{A}^\perp)$" means that a prefix of $\mathbf{Tree}(\mathcal{G})$ is invalid (hence so is $\mathbf{Tree}(\mathcal{G})$). Note that the co-trivial model checking of $\mathcal{G}$ with respect to $\mathcal{A}$ is equivalent to the trivial model checking of $\mathcal{G}$ with respect to $\overline{\mathcal{A}}$ (where $\overline{\mathcal{A}}$ is the complement of $\mathcal{A}$) considered in [1, 9, 3].

Example 3. Recall $\mathcal{G}_0$ in Example 1 and $\mathcal{A}_0$ in Example 2. Then, $\mathcal{G}_0 \not\models \mathcal{A}_0$ holds. In other words, every finite path (that ends in c) $\mathbf{Tree}(\mathcal{G})$ contains an even number of b's.

2.2 Broadbent and Kobayashi's Algorithm

We quickly review Broadbent and Kobayashi's saturation-based algorithm HorSat for co-trivial automata model checking of HORS [3]. We fix an ATA $\mathcal{A} = (\Sigma, Q, \delta, q_I)$ and a HORS $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ in the following discussion.

Definition 4 (Intersection types). *The sets **ITypes** and **STypes** of **intersection types** and **strict types**, ranged over by σ and θ respectively, are defined by:*

$$\sigma ::= \{ \theta_1, \ldots, \theta_n \} \qquad \theta ::= q \mid \bigwedge \sigma \to \theta$$

Here $q \in Q$ and n is a non-negative integer.

Intuitively, the type q describes trees accepted by the automaton from state q, and $\bigwedge \sigma \to \theta$ describes functions that take an argument that has all (strict) types in σ and returns a value of type θ.

We say θ is a **refinement** of κ, written $\theta :: \kappa$, when it is derivable by the following rules.

$$\frac{}{q :: \mathtt{o}} \qquad \frac{\sigma :: \kappa_1 \qquad \theta :: \kappa_2}{(\bigwedge \sigma \to \theta) :: (\kappa_1 \to \kappa_2)} \qquad \frac{\theta :: \kappa \text{ for each } \theta \in \sigma}{\sigma :: \kappa}$$

An **(intersection) type environment** is a map $\Gamma : \mathrm{dom}(\mathcal{N}) \to \mathbf{ITypes}$ such that $\forall f \in \mathrm{dom}(\mathcal{N}).\ \Gamma(f) :: \mathcal{N}(f)$. The union of type environments $\Gamma_1 \cup \Gamma_2$ is defined by $(\Gamma_1 \cup \Gamma_2)(x) = \Gamma_1(x) \cup \Gamma_2(x)$.

The type judgment relation $\Gamma \vdash_I t : \theta$ is defined by the following typing rules:

$$\frac{\theta \in \Gamma(f)}{\Gamma \vdash_I f : \theta} \qquad \frac{(q, a, U) \in \delta}{\Gamma \vdash_I a : \bigwedge U|_1 \to \cdots \to \bigwedge U|_{\mathsf{arity}(\Sigma(a))} \to q}$$

$$\frac{\Gamma \ \vdash_I \ t_1 : \bigwedge \sigma \to \theta \qquad \Gamma \ \vdash_I \ t_2 : \theta' \text{ for each } \theta' \in \sigma}{\Gamma \ \vdash_I \ t_1\ t_2 : \theta}$$

Here, $U|_i = \{\, q \mid (j, q) \in U, j = i \,\}$.

A type environment Γ can be considered a finite representation of the set of terms: $\mathbf{ITerms}_{\Gamma,q_I} = \{\, t \mid \Gamma \vdash_I t : q_I \,\}$. The set $\mathbf{ITerms}_{\emptyset,q_I}$ described by the empty type environment is exactly the set of terms t such that $t^{\perp} \in \mathcal{L}(\mathcal{A}^{\perp})$. HorSat starts from the empty type environment, and iteratively expand it to obtain a type environment Γ such that $\mathbf{ITerms}_{\Gamma,q_I} = \{\, t \mid \exists s. t \longrightarrow^* s \wedge s^{\perp} \in \mathcal{L}(\mathcal{A}^{\perp}) \,\} \cap$ **RTerms**, where **RTerms** is an over-approximation of the set of terms reachable from S, i.e., $\{\, t \mid S \longrightarrow^* t \,\}$. Once such Γ is obtained, the co-trivial model checking amounts to checking whether $\Gamma \vdash_I S : q_I$, i.e., whether $q_I \in \Gamma(S)$ holds.

HorSat makes use of flow information to efficiently compute Γ above.

Definition 5. $\mathbf{Flow}_{ap} : \mathrm{dom}(\mathcal{X}) \to \mathcal{P}(\mathbf{Terms}_{\Sigma\cup\mathcal{N}})$ *(recall that $\mathcal{X}$ is a sorted alphabet of variables in $\mathcal{G}$) is (approximate)* flow information, *if* $\mathbf{Flow}_{ap}(x_i) \subseteq \mathbf{Terms}_{\Sigma\cup\mathcal{N},\mathcal{X}(x_i)}$ *and if $t_i \in \mathbf{Flow}_{ap}(x_i)$ holds for each $i \in \{\, 1, \ldots, k \,\}$ whenever $S \longrightarrow^* t$ and $F\, t_1 \cdots t_k$ occurs as a subterm of t with $F\, x_1 \cdots x_k \to t \in \mathcal{R}$.*

Using $\mathbf{Flow}_{ap}$, the function to iteratively expand a type environment is defined as follows.

Definition 6. *The function $\mathcal{F}_{\mathcal{G}}$ over type environments is given by:*

$$\mathcal{F}_{\mathcal{G}}(\Gamma)(F) = \Gamma(F) \cup \left\{ \bigwedge \Delta(x_1) \to \cdots \to \bigwedge \Delta(x_n) \to q \;\middle|\; \begin{array}{c} F\, x_1 \cdots x_n \to t \in \mathcal{R}, \\ \Gamma \ \vdash_I \ t : q \Longrightarrow \Delta, \\ \mathbf{Inhabited}(\Gamma, \Delta) \end{array} \right\}. \tag{1}$$

Here, $\mathbf{Inhabited}(\Gamma, \Delta) \iff \forall x \in \mathrm{dom}(\Delta).\ \exists t \in \mathbf{Flow}_{ap}(x). \forall \theta \in \Delta(x).\ \Gamma \ \vdash_I t : \theta$. *The relation $\Gamma \vdash_I t : \theta \Longrightarrow \Delta$ is defined by:*

$$\frac{\Gamma \ \vdash_I \ t : \theta}{\Gamma \ \vdash_I \ t : \theta \Longrightarrow \emptyset} \qquad \frac{\exists t \in \mathbf{Flow}_{ap}(x).\ \Gamma \ \vdash_I \ t : \theta}{\Gamma \ \vdash_I \ x : \theta \Longrightarrow \{\, x : \theta \,\}}$$

$$\frac{\Gamma \ \vdash_I \ t_1 : \theta_1 \wedge \cdots \wedge \theta_n \to \theta \Longrightarrow \Delta_0 \qquad \forall i \in \{\, 1, \ldots, n \,\}.\ \Gamma \ \vdash_I \ t_2 : \theta_i \Longrightarrow \Delta_i}{\Gamma \ \vdash_I \ t_1\ t_2 : \theta \Longrightarrow \bigcup_{i=0}^{n} \Delta_i}$$

The rules for $\Gamma \vdash_I t : \theta \Longrightarrow \Delta$ can be read as an algorithm to compute Δ such that $\Gamma \cup \Delta \vdash_I t : \theta$. For example, the rule for $t_1 t_2$ says that given Γ and θ, we should first enumerate all the pairs $(\theta_1 \wedge \cdots \wedge \theta_n, \Delta_0)$ such that $\Gamma \vdash_I t_1 : \theta_1 \wedge \cdots \wedge \theta_n \to \theta \Longrightarrow \Delta_0$, and then for each such pair, enumerate all $(\Delta_1, \ldots, \Delta_n)$ such that $\Gamma \vdash_I t_2 : \theta_i \Longrightarrow \Delta_i$, and return $\bigcup_{i=0}^{n} \Delta_i$ for all the combinations of $\Delta_0, \Delta_1, \ldots, \Delta_n$.

HorSat is based on the following theorem, and computes $\Gamma = \bigcup_{i\in\omega} \mathcal{F}_{\mathcal{G}}^{i}(\emptyset)$ and checks whether $q_I \in \Gamma(S)$ holds.

Theorem 1 ([3]). *Let* $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ *be a HORS.* $q_I \in (\bigcup_{i\in\omega} \mathcal{F}_{\mathcal{G}}^{i}(\emptyset))(S)$ *if and only if* $\mathcal{G} \models \mathcal{A}$.

Example 4. Recall $\mathcal{G}_0$ in Example 1 and $\mathcal{A}_0$ in Example 2. The map

$$\{\, f \mapsto \{\, T^k \mathtt{b} \mid k \geq 1 \,\}, g \mapsto \{\, T^k \mathtt{b} \mid k \geq 0 \,\}, x \mapsto \{\, \mathtt{b}^k \mathtt{c} \mid k \geq 0 \,\} \,\}$$

is a valid flow map $\mathbf{Flow}_{ap}$. Since $\emptyset \vdash_I f(f\,x) : q_1 \Longrightarrow \Delta$ and $\mathbf{Inhabited}(\emptyset, \Delta)$ hold for $\Delta = \{\, f : \{\, q_0 \to q_1, q_1 \to q_0 \,\}, x : q_1 \,\}$, we have
$\mathcal{F}_{\mathcal{G}_0}(\emptyset) = \{\, S : \emptyset, F : \emptyset, T : \{\, (q_0 \to q_1) \wedge (q_1 \to q_0) \to q_1 \to q_1 \,\} \,\}$ and
$(\bigcup_{i\in\omega} \mathcal{F}_{\mathcal{G}}^{i}(\emptyset))(S) = \{\, q_1 \,\}$. Thus, we have $\mathcal{G}_0 \not\models \mathcal{A}_0$.

3 A ZDD-based Algorithm

We now discuss our new algorithm. The main limitations of HorSat and our approach to address them are summarized as follows.

1. First, although $\bigcup_{i\in\omega} \mathcal{F}_{\mathcal{G}}^{i}(\emptyset)$ is guaranteed to be finite, it sometimes becomes quite large, containing "similar" types $q_1 \to q_3 \to q$, $q_2 \to q_3 \to q$, $q_1 \to q_4 \to q$, and $q_2 \to q_4 \to q$, which could be represented by a single type $q_1 \vee q_2 \to q_3 \vee q_4 \to q$ if we had union types as well. The blow-up of the size of a type environment also significantly affects the cost of intermediate computation of $\mathcal{F}_{\mathcal{G}}(\Gamma)(F)$, as we have to enumerate Δ such that $\Gamma \vdash_I t : q \Longrightarrow \Delta$ and $\mathbf{Inhabited}(\Gamma, \Delta)$ one by one, and construct new types. This suggests that the intersection types (in the syntactic representation) may not be an optimal representation for computing $\bigcup_{i\in\omega} \mathcal{F}_{\mathcal{G}}^{i}(\emptyset)$. We use *ZDD* to represent intersection types and type environments, and re-define $\mathcal{F}_{\mathcal{G}}$ accordingly.

2. Secondly, HorSat uses 0CFA to compute approximate flow information $\mathbf{Flow}_{ap}$, whose worst-case complexity is almost cubic time [16]. As the definition of $\mathcal{F}_{\mathcal{G}}$ suggests, however, what we actually need is not $\mathbf{Flow}_{ap}$ itself but the set $\{\, \{\, \theta \mid \Gamma \vdash_I t : \theta \,\} \mid t \in \mathbf{Flow}_{ap}(x) \,\}$ (for each x). The latter can be more efficiently computed (in fact, in *linear time* under a fixed-parameter assumption) by first computing *sub-transitive* flow information [5] and then directly computing the set $\{\, \{\, \theta \mid \Gamma \vdash_I t : \theta \,\} \mid t \in \mathbf{Flow}_{ap}(x) \,\}$ using the sub-transitive flow information.

We discuss the first issue in Sections 3.1 and 3.2, and the second issue in Section 3.3.

3.1 ZDD types

We use ZDD [17] to represent a set of intersection types compactly. ZDD is an efficient data structure for representing a set of (finite) sets. The following description is actually based on set operations, and not tied to the specific data structure of ZDD; thus one may use other representations such as ordered boolean decision diagrams (OBDD) and boolean formulas to implement the algorithm below. Using ZDD, however, we expect that the representation is more compact and the set operations can be efficiently performed: see Remark 1 below.

We first modify the representation of a strict type.

Definition 7 (ZDD types). *Let θ be a strict type. The **ZDD strict type** corresponding to θ, written $[\theta]$, is defined as:*

$$[q] = \{\, q \,\} \qquad [\textstyle\bigwedge \sigma \to \theta] = \{\, (\mathsf{arity}(\textstyle\bigwedge \sigma \to \theta), \theta') \mid \theta' \in \sigma \,\} \cup [\theta]$$

*A **ZDD intersection type** is a collection of ZDD strict types. The set of ZDD intersection types is written* $\mathbf{ITypes}_{ZDD}$*. Let Γ be an intersection type environment, The **ZDD type environment** corresponding to Γ, written $[\Gamma]$, is the map from* $\mathrm{dom}(\Gamma)$ *to ZDD intersection types such that for each* $F \in \mathrm{dom}(\Gamma)$, $[\Gamma](F) = \{\, [\theta] \mid \theta \in \Gamma(F) \,\}$.

For example, the strict type $q_1 \wedge q_2 \to q_3 \to q$ is now expressed by the set $\{\, (2, q_1), (2, q_2), (1, q_3), q \,\}$.[2] The intersection type (or, the set of strict types):

$$\{\, q_1 \to q_3 \to q, q_1 \to q_4 \to q, q_2 \to q_3 \to q, q_2 \to q_4 \to q \,\}$$

is expressed by a set of sets:

$$\{\, \{\, (2, q_1), (1, q_3), q \,\}, \{\, (2, q_1), (1, q_4), q \,\}, \{\, (2, q_2), (1, q_3), q \,\}, \{\, (2, q_2), (1, q_4), q \,\} \,\}\,.$$

A careful reader will notice that we can then use a compact representation like $((2, q_1) \vee (2, q_2)) \wedge ((1, q_3) \vee (1, q_4)) \wedge q$ to represent the intersection type. Note that the set representation is *not* nested. For example, $(q_1 \wedge q_2 \to q) \wedge (q_3 \to q) \to q$ is expressed by: $\{\, (1, q_1 \wedge q_2 \to q), (1, q_3 \to q), q \,\}$. The strict types $q_1 \wedge q_2 \to q$ and $q_3 \to q$ are lazily converted to ZDD strict types as necessary inside the algorithm described below. We use meta-variables $\overline{\theta}$, $\overline{\sigma}$, and $\overline{\Gamma}$ for ZDD strict types, ZDD intersection types and ZDD type environments respectively. (In general, we shall use $\overline{x}$ as the meta-variable for the ZDD version of x below.)

In the saturation-based algorithm (recall Definition 6), we need to compute the set of pairs (θ, Δ) such that $\Gamma \vdash_I t : \theta \Longrightarrow \Delta$ for given Γ and t. We therefore prepare a representation for such a set.

Definition 8. *Let θ be a strict type, and Δ be an intersection type environment. The **ZDD constraint type** corresponding to $\theta \Longrightarrow \Delta$, written $[\theta \Longrightarrow \Delta]$, is defined by:*

$$[\theta \Longrightarrow \Delta] = [\theta] \cup [\Delta]_s \tag{2}$$

$$[\Delta]_s = \{\, (x, \theta) \mid x \in \mathrm{dom}(\Delta), \theta \in \Delta(x) \,\} \tag{3}$$

[2] This is actually similar to the representation of Ong's *variable profiles* [19]: $(\{\, (x_2, q_1), (x_2, q_2), (x_1, q_3) \,\}, q)$.

We use the meta-variable $\overline{\Delta}$ for a subset of $\mathcal{X} \times \mathbf{STypes}$, and τ for a ZDD constraint type. Please notice the difference between $[\Gamma]$ and $[\Delta]_s$. In the former, types are converted to ZDD ones, while in the latter, types are kept as they are. For example, a constraint strict type $q_1 \to q \Longrightarrow \{f : \{q_1 \to q_2\}, x : \{q_1, q_2\}\}$ is represented as a ZDD constraint type $\{q, (1, q_1), (f, q_1 \to q_2), (x, q_1), (x, q_2)\}$. Since ZDD strict type is a subset of $Q \cup (\mathbb{N} \times \mathbf{STypes})$, and a ZDD constraint type is a subset of $Q \cup (\mathbb{N} \times \mathbf{STypes}) \cup (\mathcal{X} \times \mathbf{STypes})$, a collection of them can be represented using ZDD, by treating elements of $Q, \mathbb{N} \times \mathbf{STypes}, \mathcal{X} \times \mathbf{STypes}$ as atomic elements.

Let τ be a collection of ZDD constraint types, $q \in Q$, Θ be a subset of $\mathbb{N} \times \mathbf{STypes}$ and $\overline{\Delta}$ be a subset of $\mathcal{X} \times \mathbf{STypes}$. We write $(q, \Theta, \overline{\Delta}) \in \tau$ when $\{q\} \cup \Theta \cup \overline{\Delta} \in \tau$. We write $(q, \Theta) \in \overline{\sigma}$ when $\overline{\sigma}$ is a ZDD intersection type and $\{q\} \cup \Theta \in \overline{\sigma}$. The notation $\Theta(i)$ and $\overline{\Delta}(x)$ respectively denote the sets $\{\, \theta \mid (i, \theta) \in \Theta \,\}$ and $\{\, \theta \mid (x, \theta) \in \Delta \,\}$ (which is based on the standard set representation of a map).

We define a conversion from ZDD intersection types to intersection types.

Definition 9. *Let $\overline{\sigma}$ be a ZDD intersection type, and n be a non-negative integer. The intersection type corresponding to $\overline{\sigma}$ with the arity n, written* $\mathsf{enum}(\tau, n)$ *is defined by:*

$$\mathsf{enum}(\overline{\sigma}, n) = \left\{ \bigwedge \Theta(n) \to \cdots \to \bigwedge \Theta(1) \to q \;\middle|\; (q, \Theta) \in \overline{\sigma} \right\} \tag{4}$$

3.2 Saturation algorithm using ZDD types

We now present the new saturation-based algorithm using ZDD types.

Definition 10. *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ be a HORS. The function $\overline{\mathcal{F}}_{\mathcal{G}}$ over ZDD type environments is defined by:*

$$\overline{\mathcal{F}}(\overline{\Gamma})(F) = \overline{\Gamma}(F) \cup \left\{ \mathsf{rename}(\mathsf{inhabited}(\tau, \overline{\Gamma})) \;\middle|\; \begin{array}{c} F\, x_n \cdots x_1 \to t \in \mathcal{R}, \\ \overline{\Gamma} \vdash_{ZDD} t : \tau \end{array} \right\} \tag{5}$$

Here, $\mathsf{rename}(\tau)$*,* $\mathsf{inhabited}(\tau, \overline{\Gamma})$*, and $\overline{\Gamma} \vdash_{ZDD} t : \tau$ are defined by:*

$$\begin{array}{l} \mathsf{rename}(\tau) = \{\, \{q\} \cup \{\, (i, \theta) \mid (x_i, \theta) \in \overline{\Delta} \,\} \mid (q, \emptyset, \overline{\Delta}) \in \tau \,\} \\ \mathsf{inhabited}(\tau, \overline{\Gamma}) = \{\{q\} \cup \overline{\Delta} | (q, \emptyset, \overline{\Delta}) \in \tau, \quad \exists \overline{\sigma}. \overline{\Delta}(x) \subseteq \overline{\sigma} \wedge \overline{\sigma} \in \mathtt{typesof}(x, \overline{\Gamma})\} \end{array}$$

$$\frac{F \in \mathrm{dom}(\mathcal{N})}{\overline{\Gamma} \;\vdash_{ZDD}\; F : \overline{\Gamma}(F)}$$

$$\frac{a \in \mathrm{dom}(\Sigma)}{\overline{\Gamma} \;\vdash_{ZDD}\; a : \left\{ \{q\} \cup \left\{ (k, q') \;\middle|\; \begin{array}{c} (j, q') \in U \\ k = \mathsf{arity}(\Sigma(a)) - j + 1 \end{array} \right\} \;\middle|\; \begin{array}{c} q \in Q, \\ (q, a, U) \in \delta \end{array} \right\}}$$

$$\frac{x \in \mathrm{dom}(\mathcal{X})}{\overline{\Gamma} \;\vdash_{ZDD}\; x : \{\, [\theta] \cup \{(x, \theta)\} \mid \theta \in \bigcup \mathtt{typesof}(x, \overline{\Gamma}) \,\}}$$

$$\frac{\overline{\Gamma} \vdash_{ZDD} t_1 : \tau_1 \qquad \overline{\Gamma} \vdash_{ZDD} t_2 : \tau_2 \qquad n = \mathsf{arity}(t_1)}{\overline{\Gamma} \vdash_{ZDD} t_1\ t_2 : \left\{ \{q\} \cup \Theta' \cup \overline{\Delta} \cup \overline{\Delta}' \;\middle|\; \begin{array}{c} (q, \Theta, \overline{\Delta}) \in \tau_1 \\ \overline{\Delta}' \in \bigotimes_{\theta \in \Theta(n)} g(\tau_2, \theta) \\ \Theta' = \Theta \setminus \{\, (n, \theta) \mid \theta \in \Theta(n) \,\} \end{array} \right\}}$$

where $S_1 \otimes S_2 = \{\, s_1 \cup s_2 \mid s_1 \in S_1, s_2 \in S_2 \,\}$, $\Theta(n) = \{\, (j, \theta) \mid (j, \theta) \in \Theta, n = j \,\}$, *and* $g(\tau, \theta) = \{\, \overline{\Delta} \mid (q, \Theta, \overline{\Delta}) \in \tau, [\theta] = \{q\} \cup \Theta \,\}$.
$\mathtt{typesof}(x, \overline{\Gamma}) = \{\, \mathsf{enum}(\overline{\sigma}_{\overline{\Gamma},t}, \mathsf{arity}(\mathcal{X}(x))) \mid t \in \mathbf{Flow}_{ap}(x) \,\}$ *where* $\overline{\sigma}_{\overline{\Gamma},t}$ *is the (unique) intersection type such that* $\overline{\Gamma} \vdash_{ZDD} t{:}\overline{\sigma}_{\overline{\Gamma},t}$. *(Since* t *is closed,* τ *such that* $\overline{\Gamma} \vdash_{ZDD} t : \tau$ *contains no free variables, hence it is actually a ZDD intersection type.)*

Note that the relation $\Gamma \vdash_I t : \theta \Longrightarrow \Delta$ has now been replaced by $\overline{\Gamma} \vdash_{ZDD} t : \tau$. Since τ represents a set of pairs $(\theta_1, \Delta_1), \ldots, (\theta_n, \Delta_n)$, $\overline{\Gamma} \vdash_I t : \tau$ means that $\overline{\Gamma} \vdash_I t : \theta_i \Longrightarrow \Delta_i$ holds for all such pairs. Thanks to this modification, the algorithm to compute τ such that $\overline{\Gamma} \vdash_I t : \tau$ is deterministic, and implemented by using ZDD. The set $\mathtt{typesof}(x, \overline{\Gamma})$ used above is based on flow information $\mathbf{Flow}_{ap}$. How to represent $\mathbf{Flow}_{ap}$ and compute $\mathtt{typesof}(x, \overline{\Gamma})$ using it is explained later in Section 3.3.

The following lemma formally states the correspondence between $\Gamma \vdash_I t : \theta \Longrightarrow \Delta$ and $\overline{\Gamma} \vdash_{ZDD} t : \tau$ mentioned above.

Lemma 1. *Let* Γ *be an intersection type environment over* $\mathcal{N}$, t *be an applicative term. For any* θ *and* Δ, *if* $\Gamma \ \vdash_I \ t : \theta \Longrightarrow \Delta$ *then there exists* τ *such that* $[\Gamma] \vdash_{ZDD} t : \tau$ *and* $[\theta \Longrightarrow \Delta] \in \tau$. *Conversely, for any* φ *and* τ, *if* $[\Gamma] \vdash_{ZDD} t : \tau$ *and* $\varphi \in \tau$, *there exist* θ *and* Δ *such that* $\Gamma \ \vdash_I \ t : \theta \Longrightarrow \Delta$ *and* $\varphi = [\theta \Longrightarrow \Delta]$.

Based on the above lemma, we can obtain the following correspondence between the step functions used for saturation.

Lemma 2. *Let* $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ *be a HORS and* Γ *be an intersection type environment over* $\mathcal{N}$. *Then the following equation holds.*

$$\overline{\mathcal{F}}_{\mathcal{G}}([\Gamma]) = [\mathcal{F}_{\mathcal{G}}(\Gamma)] \tag{6}$$

The following theorem is an immediate corollary of Theorem 1 and Lemma 2.

Theorem 2. *Let* $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ *be a HORS.* $q_I \in \bigcup_{i \in \omega} (\overline{\mathcal{F}}^i_{\mathcal{G}}(\emptyset))(S)$ *if and only if* $\mathcal{G} \models \mathcal{A}$.

Remark 1. The formalization above does not rely on the specific data structure of ZDD [17]. We could, therefore, use OBDD instead. In fact, our initial implementation used OBDD rather than ZDD. According to our earlier experiments, however, ZDD tends to be more efficient. Our rationale for this is that in many of the benchmarks, while the “width” of each intersection type (i.e., the size σ) tends to be small, the number of strict types θ that occur in a set of intersection types can be large. Due to this property, suppressing zero’s in ZDD brings a benefit. This argument is however yet to be confirmed through more experiments.

3.3 Approximation of Control-flow information

Next, we discuss how to compute $\mathbf{Flow}_{ap}$ and $\mathtt{typesof}(x, \overline{\Gamma})$ efficiently. To obtain (a finite representation of) $\mathbf{Flow}_{ap}$, we use Heintze and Mcallester's sub-transitive flow analysis [5].

Definition 11 (Sub-transitive flow graph). *Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ be a HORS. A **sub-transitive** flow graph of $\mathcal{G}$ is a quadruple (V, E, ξ, ρ) such that: (i) (V, E) is a directed acyclic graph, (ii) each leaf v in V is labeled by $\xi(v) \in \mathbf{Terms}_{\Sigma \cup \mathcal{N} \cup \mathcal{X}}$, and (iii) $\rho : \mathrm{dom}(\mathcal{X}) \to V$. The **flow map** represented by a sub-transitive flow graph (V, E, ξ, ρ) is the least (with respect to the pointwise ordering) map $h : \mathrm{dom}(\mathcal{X}) \to \mathbf{Terms}_{\Sigma, \mathcal{N}}$ such that*

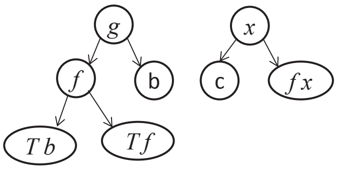
$$h(x) = \{\, t \in \mathsf{subst}(\xi(v)) \mid v \text{ is reachable from } \rho(x) \,\}\,.$$

Here, $\mathsf{subst}(t)$ is defined inductively by:

$$\mathsf{subst}(a) = \{\, a \,\} \qquad \mathsf{subst}(F) = \{\, F \,\} \qquad \mathsf{subst}(x) = h(x)$$
$$\mathsf{subst}(t_1\ t_2) = \{\, t'_1\ t'_2 \mid t'_1 \in \mathsf{subst}(t_1), t'_2 \in \mathsf{subst}(t_2) \,\}$$

A sub-transitive flow graph is sound if its flow map h is approximate flow information.

Example 5. Recall $\mathcal{G}_0$ in Example 1. A sub-transitive flow graph for $\mathcal{G}_0$ is depicted as below:



Here, the label for each non-leaf node shows the map f (e.g., x means that $\rho(x)$ is the node labeled by x in the graph), and the label for each leaf node shows the map ξ.

We can compute a sub-transitive flow graph whose flow map is equivalent to the result of 0CFA in time *linear* in the size of HORS by using Heintze and Mcallester's algorithm[5], under the assumption that the size of the largest type used in HORS is fixed. Therefore, the size of the sub-transitive flow graph is also linear in the size of HORS. We use it as a finite representation of $\mathbf{Flow}_{ap}$ below.

Let $G = (V, E, \xi, \rho)$ be a sound sub-transitive flow graph. Let $\overline{\Gamma}$ be a ZDD type environment over $\mathcal{N}$. We present an algorithm to compute $\mathtt{typesof}(x, \overline{\Gamma})$. We define the function $\mathcal{H}_{\overline{\Gamma}} : (\mathrm{dom}(\mathcal{X}) \to \mathcal{P}(\mathbf{ITypes}_{ZDD})) \to (\mathrm{dom}(\mathcal{X}) \to \mathcal{P}(\mathbf{ITypes}_{ZDD}))$ by:

$$\mathcal{H}_{\overline{\Gamma}}(\Xi)(x) = \Xi(x) \cup \{\, \overline{\sigma} \mid \ v \text{ is reachable from } \rho(x), \overline{\Gamma}; \Xi \vdash_{ZDD} \xi(v) : \overline{\sigma} \,\} \qquad (7)$$

where $\overline{\Gamma}; \Xi \vdash_{ZDD} t : \overline{\sigma}$ is given by:

$$\frac{f \in \mathrm{dom}(\mathcal{N}) \cup \mathrm{dom}(\Sigma) \qquad \overline{\Gamma} \vdash_{ZDD} f : \overline{\sigma}}{\overline{\Gamma}, \Xi \vdash_{ZDD} f : \overline{\sigma}}$$

$$\frac{\overline{\sigma} \in \Xi(x)}{\overline{\Gamma}, \Xi \vdash_{ZDD} x : \overline{\sigma}}$$

$$\frac{\overline{\Gamma}, \Xi \vdash_{ZDD} t_1 : \overline{\sigma}_1 \qquad \overline{\Gamma}, \Xi \vdash_{ZDD} t_2 : \overline{\sigma}_2 \qquad n = \mathsf{arity}(t_1)}{\overline{\Gamma}, \Xi \vdash_{ZDD} t_1\ t_2 : \left\{ \{q\} \cup \Theta' \;\middle|\; \begin{array}{c} (q, \Theta) \in \overline{\sigma}_1 \\ \Theta(n) \subseteq \mathsf{enum}(\overline{\sigma}_2, \mathsf{arity}(t_2)) \\ \Theta' = \Theta \setminus \{\, (n, \theta) \mid \theta \in \Theta(n) \,\} \end{array} \right\}}$$

Lemma 3. *Let* $\Xi_0 = \{\, x : \emptyset \mid x \in \mathrm{dom}(\mathcal{X}) \,\}$*, and* $\Xi^{(\omega)} = \bigcup_{n \in \omega} (\mathcal{H}_{\overline{\Gamma}})^n (\Xi_0)$*, and* $x \in \mathrm{dom}(\mathcal{X})$*.* $\forall t \in \mathbf{Flow}_{ap}(x). \exists \overline{\sigma} \in \Xi^{(\omega)}.\ \overline{\Gamma} \vdash_{ZDD} t : \overline{\sigma}$*, and* $\forall \overline{\sigma} \in \Xi^{(\omega)}(x).\ \exists t \in \mathbf{Flow}_{ap}(x).\ \overline{\Gamma} \vdash_{ZDD} t : \overline{\sigma}$*.*

Because the number of all intersection types are finite, we can compute $\Xi^{(\omega)}$ and use it to compute $\mathcal{F}_{\mathcal{G}}(\overline{\Gamma})$.

3.4 Fixed-Parameter Linear Time Algorithm

We now discuss how to compute $\bigcup_{i \in \omega} (\overline{\mathcal{F}}^{i}_{\mathcal{G}}(\emptyset))$ (recall Theorem 2) in time linear in the size of HORS, under the assumption that (i) the largest order and size of types in HORS and (ii) the property automaton $\mathcal{A}$ are fixed. This fixed-parameter assumption is the same as the assumption made in the literature [9, 10, 8, 3, 21]. The naive fixed computation of $\bigcup_{i \in \omega} (\overline{\mathcal{F}}^{i}_{\mathcal{G}}(\emptyset))$ and $\bigcup_{n \in \omega} (\mathcal{H}_{\overline{\Gamma}})^n (\Xi_0)$ is polynomial time, but not linear: both the number of iterations to compute $\overline{\mathcal{F}}_{\mathcal{G}}(\overline{\Gamma})$ and the cost for each iteration are linear in the size of HORS even if we assume $\bigcup_{n \in \omega} (\mathcal{H}_{\overline{\Gamma}})^n (\Xi_0)$ can be computed in linear time. We can, however, use the standard technique for optimizing a fixed-point computation over a finite semi-lattice [22], as follows.

We compute the following information incrementally.

- For each non-terminal F, $\overline{\Gamma}(F)$.
- For each sub-term t of the right-hand side of a rule, τ_t such that $\overline{\Gamma} \vdash_{ZDD} t : \tau_t$ (for the current value of $\overline{\Gamma}$ and `typesof`).
- For each node v of the sub-transitive flow graph (V, E, ξ, ρ), the set U_v of (ZDD) intersection types that may be taken by the term $\xi(v)$. (Without loss of generality, we assume here that for each subterm t of the right-hand side of a rule, there is a node v such that $\xi(v) = t$.)

The values $\overline{\Gamma}(F)$, τ_t and U_v are updated in an on-demand manner when other values have been updated.

- $\overline{\Gamma}(F)$ is recomputed and updated as necessary, when τ_t for the body t of F's rule or U_v such that $\rho(x_i) = v$ (where x_i is a formal parameter of F) has been updated,

- τ_x is recomputed and updated as necessary, when U_v such that $\rho(x) = v$ has been updated.
- τ_a is updated only initially.
- τ_F is recomputed and updated as necessary, when $\overline{\Gamma}(F)$ has been updated.
- $\tau_{t_1 t_2}$ is recomputed and updated as necessary, when τ_{t_1} or τ_{t_2} has been updated.
- U_v is recomputed and updated as necessary, when (i) $U_{v'}$ such that $(v, v') \in E$ has been updated, (ii) $\xi(v) = F$ and τ_F has been updated, or (iii) $\xi(v) = t_1 t_2$ and $U_{v'}$ such that $\xi(v') \in \{ t_1, t_2 \}$ has been updated.

Since each update monotonically increases the values of $\overline{\Gamma}(F)$, τ_t, and U_v (which range over finite sets), the termination is guaranteed. Under the fixed-parameter assumption, the size of the sets ranged over by $\overline{\Gamma}(F)$, τ_t, and U_v is bounded above by a constant. Thus, each recomputation and update can be performed in a constant time. The number of recomputations is linearly bounded by the size of HORS and the size of the subtransitive flow graph, where the latter is linear in the size of HORS. Thus, the whole algorithm runs in time linear in the size of HORS under the fixed-parameter assumption.

Example 6. Recall $\mathcal{G}_0$ in Example 1 and $\mathcal{A}_0$ in Example 2. After saturation, U_v and $\overline{\Gamma}$ are

$$\begin{aligned}
&U_{\rho(x)} = \{ \{ q_0 \}, \{ q_1 \} \} \\
&U_{\rho(g)} = \{ \{ q_0 \to q_1, q_1 \to q_0 \}, \{ q_0 \to q_0, q_1 \to q_1 \} \} \\
&\overline{\Gamma}(T) = \{(q_0 \to q_1) \wedge (q_1 \to q_0) \to q_0 \to q_0, (q_0 \to q_1) \wedge (q_1 \to q_0) \to q_1 \to q_1, \\
&\qquad\qquad (q_0 \to q_0) \to q_0 \to q_0, (q_1 \to q_1) \to q_1 \to q_1\} \\
&\overline{\Gamma}(F) = \{ (q_1 \to q_1) \to q_1 \} \qquad \overline{\Gamma}(S) = \{ q_1 \}
\end{aligned}$$

For readability, we wrote types in the non-ZDD notation.

4 Experiments

4.1 Data sets and evaluation environment

We have implemented our ZDD-based algorithm in the tool named HorSatZDD, evaluated its performance by existing problem instances, and compared the results with the two state-of-the-art previous higher-order model checkers: HorSat [3] and Preface [21].

The problem instances used in the benchmark are classified into three categories. The first one consists of two families of HORS, $\mathcal{G}_{m,n}$ [8] and t_n [21]. They are parametrized by m, n and have been used to evaluate the scalability of Preface [21]. The second one consists of instances automatically generated by program verification tools such as the HMTT verification tool [13], MoCHi [12], PMRS model checker [20], and exact control flow analysis [24]. They have also been used in the benchmarks for HorSat [3] and Preface [21]. The third one consists of new instances added to clarify the advantages of the new algorithm. They are also parametrized by a size parameter.

We conducted the experiments on a computer with 2.3GHz Intel Core i7 CPU, 16GB RAM and OSX 10.9.3 operating system. HorSatZDD is written in Haskell, and compiled with GHC 7.8.2. HorSat was compiled with ocamlopt version 4.01.0, and Preface was run on Mono JIT compiler version 3.2.4.

4.2 Experimental results

Figure 1 and Table 1 show the results of our experiments. In each table, columns D, S, O, and Q represent the expected decision (Y means Yes, N means No), the size of HORS (the number of occurrences of symbols in the righthand side of the rewriting rules), the order of HORS, and the number of states of automaton respectively, and the other columns represent the running time of each model checker measured in seconds.

For the instances $\mathcal{G}_{m,n}$ HorSatZDD scaled almost linearly with respect to the grammar size n. HorSatZDD scaled better than the other model checkers with respect to the grammar order m, although the running time was exponential in the grammar order due to the explosion of the sub-transitive flow graph. For the t_n instances Preface did not scale well (as reported in [21]), while both HorSatZDD and HorSat scaled well.

HorSatZDD processed all instances in the category 2 within the time limit. HorSatZDD ran in ten seconds for most test cases in the category 2, but HorSatZDD is significantly slower than the other model checkers for the instances xhtmlf-div-2, xhtmlf-m-church, jwig-cal_main, and cfa-life2. Except for cfa-life2, this is attributed to the size of the property automaton, which blows up the size of each ZDD. This suggests that further optimization of ZDD implementation is required. As for cfa-life2, the majority of the running time of HorSatZDD was for the computation of the sub-transitive flow graph. This suggests that a further optimization may be necessary on the construction of sub-transitive flow graphs.

Category 3 consists of two families of problem instances: ae3-n and abc-len$n$. The family ae3-n has been manually constructed to clarify the advantage of using ZDD to represent (a set of) intersection types. It consists of the following grammar:

$$
\begin{array}{l}
S \rightarrow \mathtt{br}\,(F \ \underbrace{\mathtt{a}_1\ \mathtt{e}_1 \ \ \cdots \ \ \mathtt{a}_1\ \mathtt{e}_1}_{n \text{ repetitions of } \mathtt{a}_1\,\mathtt{e}_1}\ )\ (F\ \mathtt{a}_2\ \mathtt{e}_2\ \cdots\ \mathtt{a}_2\ \mathtt{e}_2\ )\ (F\ \mathtt{a}_3\ \mathtt{e}_3\ \cdots\ \mathtt{a}_3\ \mathtt{e}_3\ ) \\
F\ f_1\ x_1\ \cdots\ f_n\ x_n \rightarrow f_1(x_1(\cdots(f_n(x_n\mathtt{end}))\cdots))
\end{array}
$$

Here, the types of constants are:

$$
\begin{array}{l}
\mathtt{a}_i : q_i \rightarrow q_0, \\
\mathtt{e}_i : q_0 \rightarrow q_i \wedge \bigwedge\{\, \top \rightarrow q_j \mid j \in \{\,1,2,3\,\} \setminus \{\,i\,\}\,\} \\
\mathtt{br} : (q_0 \rightarrow \top \rightarrow \top q_0) \wedge (\top \rightarrow q_0 \rightarrow \top \rightarrow q_0) \wedge (\top \rightarrow \top \rightarrow q_0 \rightarrow q_0), \mathtt{end} : q_1 \wedge q_2
\end{array}
$$

(The point is that the composition of $\mathtt{a}_i$ and $\mathtt{e}_i$ has type $q_0 \rightarrow q_0$ for every i.) HorSat uses the naive representation of intersection types and enumerates

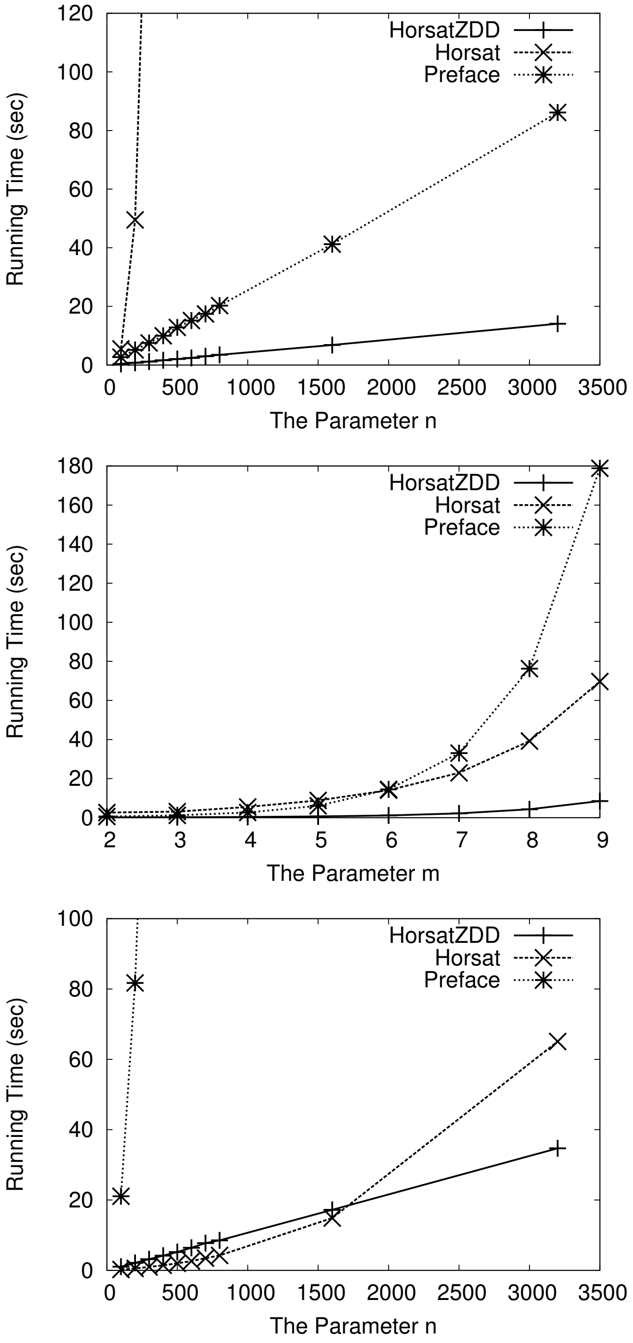


Fig. 1. Category 1: Benchmarks of $\mathcal{G}_{4,n}$ (top), $\mathcal{G}_{m.100}$ (middle), and t_n (bottom).

inputs	D	S	O	Q	ZDD	HorSat	Preface
checknz	Y	93	2	1	0.020	0.003	0.318
merge4-2	N	141	2	27	0.998	0.028	0.369
merge4	Y	141	2	27	0.906	0.031	0.519
gapid-2	Y	182	3	9	0.431	0.027	0.545
last	Y	193	2	1	0.053	0.014	0.326
checkpairs	N	251	2	1	0.055	0.018	0.379
tails	Y	259	3	1	0.063	0.021	0.331
map-plusone	Y	302	5	2	0.165	0.035	0.457
safe-head	Y	354	3	1	0.108	0.030	0.409
mc91-2	Y	358	4	1	0.222	0.060	1.934
map-head-filter	N	370	3	1	0.112	0.076	0.410
mkgroundterm	Y	379	2	1	0.103	0.042	0.347
safe-tail	Y	468	3	1	0.171	0.039	0.445
filter-nonzero	N	484	5	1	0.288	0.064	0.655
risers	Y	563	2	1	0.154	0.047	0.457
safe-init	Y	680	3	1	0.284	0.064	0.481
search-e-church	N	837	6	2	6.065	0.297	4.601
map-head-filter-1	Y	880	3	1	0.475	0.133	0.467
filter-nonzero-1	Y	890	5	2	0.887	0.159	2.357
fold_right	Y	1310	5	2	3.647	21.646	0.370
fold_fun_list	Y	1346	7	2	1.421	0.161	0.364
cfa-psdes	Y	1819	7	2	2.796	0.128	0.417
specialize_cps_coerce1-c	Y	2731	3	4	1.606	1.176	0.505
cfa-matrix-1	Y	2944	8	2	4.030	0.307	0.484
zip	Y	2952	4	2	10.425	2.276	0.916
xhtmlf-div-2	N	3003	2	50	105.414	7.846	2.024
xhtmlf-m-church	Y	3027	2	50	56.187	5.808	1.134
filepath	Y	5956	2	1	0.693	0.396	0.665
jwig-cal_main	Y	7627	2	51	73.940	7.852	0.702
cfa-life2	Y	7648	14	2	35.978	1.849	1.15
ae3-6	Y	53	2	4	0.123	0.077	0..380
ae3-8	Y	69	2	4	0.201	4.748	0.320
ae3-10	Y	85	2	4	0.312	DNF	0.309
abc-len6	Y	70	3	1	0.012	0.002	0.372
abc-len8	Y	92	3	1	0.016	0.003	1.056
abc-len10	Y	114	3	1	0.023	0.003	9.597
abc-len12	Y	136	3	1	0.029	0.004	107.766
abc-len14	Y	158	3	1	0.037	0.004	DNF

Table 1. Benchmarks of categories 2 (top) and 3 (bottom)

all the types of the form: $(q_{i_1} \to q_0) \to (q_0 \to q_{i_1}) \to \cdots (q_{i_{n-1}} \to q_0) \to (q_0 \to q_{i_{n-1}}) \to (q_{i_n} \to q_0) \to (\top \to q_{i_n}) \to q_0$ (among others) for F. Since the number of those intersection types is exponential in n, HorSat shows an exponential behavior. HorSatZDD does not suffer from the problem, since the above set of intersection types can be represented compactly. Preface works well for a different reason: it keeps binding information for all the parameters of each non-terminal together, so that it can utilize information that $f_1, \ldots, f_n$ and $x_1, \ldots, x_n$ are respectively bound to the same value for each application of F. Thus, it enumerates only types of the form: $(q_i \to q_0) \to (q_0 \to q_i) \to \cdots (q_i \to q_0) \to (q_0 \to q_i) \to q_0$. While Preface is effective for ae3-n, the use of the precise flow information causes a problem for the other instance abc-len$n$. It consists of the following rules:

$$
\begin{array}{l}
S \to F_0\, G. \\
G\, f_1 \, \cdots \, f_n \to f_1(\cdots (f_n \mathtt{e}) \cdots). \\
F_i\, f \to \mathtt{br}\, (F_{i+1}(f\, \mathtt{a}))\, (F_{i+1}(f\, \mathtt{b}))\, (F_{i+1}(f\, \mathtt{c})). \text{ (for } i = 0, \ldots, n-1\text{)} \\
F_n\, f \to f.
\end{array}
$$

Preface generates the bindings $\{ f_1 \mapsto x_1, \ldots, f_n \mapsto x_n \}$ for all $x_1, \ldots, x_n \in \{ \mathtt{a}, \mathtt{b}, \mathtt{c} \}$. Thus, Preface suffers from the exponential blow up of the size of the abstract configuration graph with respect to n. The results in Table 1 confirms the observation above. Although these examples have been artificially created, we expect that the same problems can occur in HORS generated mechanically from program verification problems.

5 Related Work

The complexity of higher-order model checking is known to be k-EXPTIME complete for order-k HORS, even when the properties are restricted to safety properties (as in the present paper) [19, 11]. Until recently, the main issue has been how to cope with this hyper-exponential worst-case complexity and construct a practical algorithm that works well for typical inputs. Kobayashi [7, 9] first developed such an algorithm. Since then, a number of other practical algorithms have been developed [8, 18, 2]. The recent development of HorSat and Preface significantly improved the scalability of higher-order model checking, and shifted the focus from how to cope with hyper-exponential complexity to how to achieve (almost) linear-time complexity to deal with thousands of lines of HORS. As already mentioned in Section 1, neither HorSat nor Preface has fully achieved it; both HorSat and Preface are fixed-parameter polynomial time algorithms (with the same fixed-parameter assumption), but HorSat suffers from cubic bottleneck of 0CFA, and Preface runs in time exponential in the largest arity of non-terminals (in other words, the order of polynomials is the largest arity): recall abc-len$n$ in Section 4. The first practical linear-time algorithm is actually due to Kobayashi [8], but because of a large constant factor, it is often slower than other algorithms such as HorSat, and Preface.

All the algorithms mentioned above are for trivial automata model checking. For more general, modal μ-calculus (or parity tree automata) model checking of HORS (as originally considered in [6] and [19]) some practical algorithms have also been developed [15, 4]. The state-of-the-art for the modal μ-calculus mode checking for HORS is, however, much behind that for trivial automata model checking. In theory, the problem still remains fixed-parameter polynomial time [10], but not linear.

Higher-order model checkers have been used as backends of various automated verification tools for higher-order programs [9, 12, 13, 20, 24, 14]. The HORS obtained in those verification tools are typically several times larger than the source programs. Being able to handle thousands of lines of HORS is, therefore, important for enabling those tools to verify large programs.

6 Conclusion

We have proposed a new saturation-based, fixed-parameter linear time algorithm for higher-order model checking and shown its effectiveness through experiments. Although it is built on Broadbent and Kobayashi's previous algorithm, we have made two important modifications that use sub-transitive flow analysis and ZDD-based representation of (a set of) intersection types. As for future work, the implementation should be improved further, as the current implementation does not exhibit the exact (fixed-parameter) linear time complexity. The use of more accurate flow information (like the one used in Preface) would improve the efficiency further, and achieving it without losing the fixed-parameter linear time complexity is also left for future work.

Acknowledgments. We would like to thank Steven Ramsay for providing the source code of Preface, and anonymous reviewers for useful comments. This work was supported by JSPS KAKENHI 23220001.

References

1. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. Logical Methods in Computer Science 3(3) (2007)
2. Broadbent, C.H., Carayol, A., Hague, M., Serre, O.: C-SHORe: a collapsible approach to higher-order verification. In: Proceedings of ICFP '13. pp. 13–24 (2013)
3. 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)
4. Fujima, K., Ito, S., Kobayashi, N.: Practical alternating parity tree automata model checking of higher-order recursion schemes. In: Proceedings of APLAS 2013. Lecture Notes in Computer Science, vol. 8301, pp. 17–32. Springer (2013)
5. Heintze, N., McAllester, D.A.: Linear-time subtransitive control flow analysis. In: Proceedings of PLDI 97. pp. 261–272 (1997)
6. Knapik, T., Niwinski, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Proceedings of FoSSaCS 2002. LNCS, vol. 2303, pp. 205–222. Springer (2002)

7. Kobayashi, N.: Model-checking higher-order functions. In: Proceedings of PPDP 2009. pp. 25–36. ACM Press (2009)
8. Kobayashi, N.: A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In: Proceedings of FoSSaCS 2011. LNCS, vol. 6604, pp. 260–274. Springer (2011)
9. Kobayashi, N.: Model checking higher-order programs. Journal of the ACM 60(3) (2013)
10. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009. pp. 179–188. IEEE Computer Society Press (2009)
11. 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)
12. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of PLDI 2011. pp. 222–233. ACM Press (2011)
13. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: Proceedings of POPL 2010. pp. 495–508. ACM Press (2010)
14. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Proceedings of ESOP 2014. Lecture Notes in Computer Science, vol. 8410, pp. 392–411. Springer (2014)
15. Lester, M.M., Neatherway, R.P., Ong, C.H.L., Ramsay, S.J.: Model checking liveness properties of higher-order functional programs. In: Proceedings of ML Workshop 2011 (2011)
16. Midtgaard, J., Horn, D.V.: Subcubic control flow analysis algorithms. Higher-Order and Symbolic Computation
17. Minato, S.: Zero-suppressed bdds for set manipulation in combinatorial problems. In: Proceedings of DAC '93. pp. 272–277 (1993)
18. Neatherway, R.P., Ramsay, S.J., Ong, C.H.L.: A traversal-based algorithm for higher-order model checking. In: ACM SIGPLAN International Conference on Functional Programming (ICFP '12). pp. 353–364 (2012)
19. 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)
20. 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 Press (2011)
21. Ramsay, S., Neatherway, R., Ong, C.H.L.: An abstraction refinement approach to higher-order model checking. In: Proceedings of POPL 2014 (2014)
22. Rehof, J., Mogensen, T.: Tractable constraints in finite semilattices. Science of Computer Programming 35(2), 191–221 (1999)
23. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie-Mellon University (May 1991)
24. Tobita, Y., Tsukada, T., Kobayashi, N.: Exact flow analysis by higher-order model checking. In: Proceedings of FLOPS 2012. LNCS, vol. 7294, pp. 275–289. Springer (2012)