Under consideration for publication in Formal Aspects of Computing

TFA: An Efficient and Precise Resolution for Virtual Method Call

Xilong Zhuo[1] and Chenyi Zhang[2]

[1]College of Information Science and Technology, Jinan University, Guangzhou, China
[2]College of Information Science and Technology, Jinan University, Guangzhou, China

Abstract. The problem of resolving virtual method and interface calls in object-oriented languages has been a long standing challenge to the program analysis community. The complexities are due to various reasons, such as increased levels of class inheritance and polymorphism in large programs. In this paper, we propose a new approach called type flow analysis that represent propagation of type information between program variables by a group of relations without the help of a heap abstraction. We prove that regarding the precision on reachability of class information to a variable, our method produces results equivalent to that one can derive from a points-to analysis. Moreover, we implement a static type analysis tool for our approach. To evaluate our method, we use instrumentation tenichque to implement a dynamic profiling type recorder. The experimental results show that our approach is more practical in both efficiency and accuracy.

Keywords: Type Analysis; Static Analysis; Method Resolving; Call Graph

1. Introduction

For object-oriented programming languages, virtual methods (or functions) are those declared in a base class but are meant to be overridden in different child classes. Statically determine a set of methods that may be invoked at a call site is important to program optimization, from results of which a subsequent optimization may reduce the cost of virtual function calls or perform method inlining if target method forms a singleton set, and one may also remove methods that are never called by any call sites, or produce a call graph which can be useful in other optimization processes.

Efficient solutions, such as Class Hierarchy Analysis (CHA) [**?**, **?**], Rapid Type Analysis (RTA) [**?**] and Variable Type Analysis (VTA) [**?**], conservatively assign each variable a set of class definitions, with relatively low precision. Alternatively, with the help of an abstract heap, one may take advantage of points-to analysis [**?**] to compute a set of object abstractions that a variable may refer to, and resolve the receiver classes in order to find associated methods at call sites.

The algorithms used by CHA, RTA and VTA are conservative, which aim to provide an efficient way to resolve calling edges, and which usually take linear-time in the size of a program, by focusing on the types that are collected at the receiver of a call site. For instance, let x be a variable of declared class A, then at a

Correspondence and offprint requests to: Chenyi Zhang. e-mail: chenyi_zhang@jnu.edu.cn

```
class A{
    A f;
    void m(){
        return this.f;
    }
}
class B extends A{}
class C extends A{}
```

```
1:  A x = new A();  //O_1
2:  B b = new B();  //O_2
3:  A y = new A();  //O_3
4:  C c = new C();  //O_4
5:  x.f = b;
6:  y.f = c;
7:  z = x.m();
```

Fig. 1. An example that compares precision on type flow in a program.

Statement	VTA fact
$A\ x = \mathsf{new}\ A()$	$x \leftarrow A$
$B\ b = \mathsf{new}\ B()$	$b \leftarrow B$
$A\ y = \mathsf{new}\ A()$	$y \leftarrow A$
$C\ c = \mathsf{new}\ C()$	$c \leftarrow C$
$x.f = b$	$A.f \leftarrow b$
$y.f = c$	$A.f \leftarrow c$
$z = x.m()$	$A.m.this \leftarrow x$ $A.m.return \leftarrow A.f$ $z \leftarrow A.m.return$

Fig. 2. VTA facts on the example

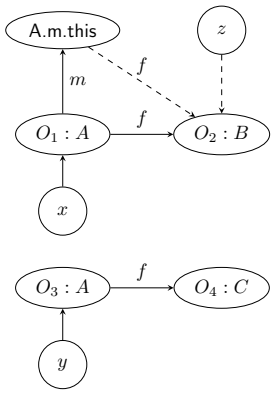


Fig. 3. Points-to results on the example

call site $x.m()$, CHA will draw a call edge from this call site to method $m()$ of class A and every definition $m()$ of a class that extends A. In case class A does not define $m()$, a call edge to an ancestor class that defines $m()$ will also be included. For a variable x of declared interface I, CHA will draw a call edge from this call site to every method of name $m()$ defined in class X that implements I. We write $CHA(x, m)$ for the set of methods that are connected from call site $x.m()$ as resolved by Class Hierarchy Analysis (CHA). Rapid Type Analysis (RTA) is an improvement from CHA which resolves call site $x.m()$ to $CHA(x, m) \cap inst(P)$, where $inst(P)$ stands for the set of methods of classes that are instantiated in the program.

Variable Type Analysis (VTA) is a further improvement. VTA defines a node for each variable, method, method parameter and field. Class names are treated as values and propagation of such values between variables work in the way of value flow. As shown in Figure 2 (example code in figure 1), the statements on line $1-4$ initialize type information for variables x, y, b and c, and statements on line $5-7$ establish value flow relations. Since both x and y are assigned type A, $x.f$ and $y.f$ are both represented by node $A.f$, thus the set of types reaching $A.f$ is now $\{B, C\}$. (Note this is a more precise result than CHA and RTA which assign $A.f$ with the set $\{A, B, C\}$.) Since $A.m.this$ refers to x, $this.f$ inside method $A.m()$ now refers to $A.f$. Therefore, through $A.m.return$, z receives $\{B, C\}$ as its final set of reaching types.

The result of a context-insensitive subset based points-to analysis [**?**] creates a heap abstraction of four objects (shown on line $1-4$ of Figure 1 as well as the ellipses in Figure 3). These abstract objects are then inter-connected via field store access defined on line $5-6$. The derived field access from $A.m.this$ to O_2 is shown in dashed arrow. By return of the method call $z = x.m()$, variable z receives O_2 of type B from $A.m.this.f$, which gives a more precise set of reaching types for variable z.

From this example, one may conclude that the imprecision of VTA in comparison with points-to analysis is due to the over abstraction of object types, such that O_1 and O_3, both of type A, are treated under the same type. Nevertheless, points-to analysis requires to construct a heap abstraction, which brings in extra information, especially when we are only interested in the set of reaching types of a variable.

In this paper we introduce a relational static semantics called Type Flow Analysis (TFA) on program variables and field accesses. Different from VTA, in addition to the binary value flow relation "$\sqsubseteq$" on the variable domain `VAR`, where $x \sqsubseteq y$ denotes all types that flow to x also flow to y, we also build a ternary

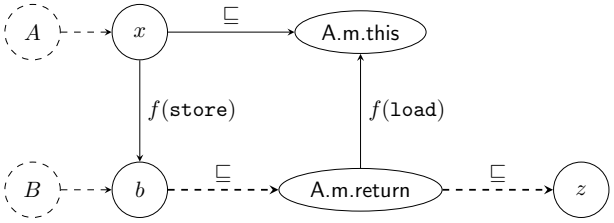


Fig. 4. Type Flow Analysis for variable z in the example

field store relation $\rightarrow \subseteq \texttt{VAR} \times \mathcal{F} \times \texttt{VAR}$ to trace the *load* and *store* relationship between variables via field accesses. This provides us additional ways to extend the relations $\sqsubseteq$ as well as $\rightarrow$. Taking the example from Figure 1, we are able to collect the store relation $x \xrightarrow{f} b$ from line 5. Since $x \sqsubseteq \mathsf{A.m.this}$, together with the implicit assignment which loads f of $\mathsf{A.m.return}$, we further derives $b \sqsubseteq \mathsf{A.m.return}$ and $b \sqsubseteq z$ (dashed thick arrows in Figure 4). Therefore, we assign type B to variable z. The complete reasoning pattern is depicted in Figure 4. Nevertheless, one cannot derive $c \sqsubseteq z$ in the same way.

We have proved that in the context-insensitive inter-procedural setting, TFA is as precise as the subset based points-to analysis regarding type related information. Since points-to analysis can be enhanced with various types of context-sensitivity on variables and objects (*e.g.* , call-site-sensitivity [**?**, **?**], object-sensitivity [**?**, **?**, **?**] and type-sensitivity [**?**]), extending type flow analysis with context-sensitivity will only require to consider contexts on variables, which is left for future work.

We have implemented a static type analysis tool for context-insensitive type flow analysis, which is written in Java and aims to analyze Java programs. We run our algorithm on the Jimple, a typed, 3-address, statemente based intermediate representation provided by Soot. The implementation has been tested on a collection of benchmark programs from SPECjvm2008 [**?**]. We conducted several experiments to compare the efficiency and accuracy with CHA and PTA implemented by Soot. In order to comparing accuracy, we used the instrument technique to implement a dynamic profiling tool, to extract the types of method receiver in runtime. We compared different static results with dynamic record. The experimental result has shown that our approach is more precise than CHA and as precise as PTA. Moreover, TFA reduce significant time cost than PTA. In addition, we found that both PTA and TFA do not provide soundness on all benchmarks. We detail looked into all the unmatch record to find out the reason for unsoundness, which involves some Java advanced features liked reflection, JNI, etc. We will discuss it in the experiment section of this paper.

2. Type Flow Analysis

We define a core calculus consisting of most of the key object-oriented language features, shown in Figure 5, which is designed in the same spirit as Featherweight Java [**?**]. A program is defined as a code base $\overline{C}$ (i.e., a collection of class definitions) with statement s to be evaluated. To run a program, one may assume that s is the default (static) entry method with local variable declarations $\overline{D}$, similar to e.g., Java and C++, which may differ in specific language designs. We define a few auxiliary functions. Let function $fields$ maps class names to their fields, $methods$ maps class names to their defined or inherited methods, and $type$ provides types (or class names) for objects. Given class c, if $f \in fields(c)$, then $ftype(c, f)$ is the defined class type of field f in c. Similarly, give an object o, if $f \in fields(type(o))$, then $o.f$ may refer to an object of type $ftype(type(o), f)$ or any of its subclass at runtime. Write $\mathcal{C}$ for the set of classes, $\texttt{OBJ}$ for the set of objects, $\mathcal{F}$ for the set of fields and $\texttt{VAR}$ for the set of variables that appear in a program. [1]

In this simple language we do not model common types (*e.g.* , int and float) that are irrelevant to our analysis. We focus on the reference types which form a class hierarchical structure. We assume a context insensitive setting, such that every variable can be uniquely determined by its name together with its enclosing class and methods. For example, if a local variable x is defined in method m of class c, then $c.m.x$ is the unique representation of that variable. Therefore, it is safe to drop the enclosing class and method name if it is clear from the context. In general, we have the following types of variables in our analysis: (1) local

[1] Sometimes we mix-use the terms *type* and *class* in this paper when it is clear from the context.

$$
\begin{array}{lll}
C & ::= & \mathsf{class}\ c\ [\mathsf{extends}\ c]\ \{\overline{F};\ \overline{M}\} \\
F & ::= & c\ f \\
D & ::= & c\ z \\
M & ::= & m(x)\ \{\overline{D}; s; \mathsf{return}\ x'\} \\
s & ::= & e \mid x{=}\mathsf{new}\ c \mid x{=}e \mid x.f{=}y \mid s; s \\
e & ::= & \mathsf{null} \mid x \mid x.f \mid x.m(y) \\
prog & ::= & \overline{C}; \overline{D}; s
\end{array}
$$

Fig. 5. Abstract syntax for the core language.

variables, (2) method parameters, (3) this reference of each method, all of which are syntactically bounded by their enclosing classes and methods.

We define three relation for catching type flow information in programs, a partial order on variables $\sqsubseteq \subseteq \texttt{VAR} \times \texttt{VAR}$, a type flow relation $\dashrightarrow \subseteq \mathcal{C} \times \texttt{VAR}$, as well as a field access relation $\longrightarrow \subseteq \texttt{VAR} \times \mathcal{F} \times \texttt{VAR}$, which are initially given as follows.

Definition 1. (Base Relations) These three basic relations represent program facts. They can be generated directly based on different statements syntactically appearing in programs.

1. $c \dashrightarrow x$ if there is a statement $x = \mathsf{new}\ c$;
2. $y \sqsubseteq x$ if there is a statement $x = y$;
3. $x \xrightarrow{f} y$ if there is a statement $x.f = y$.

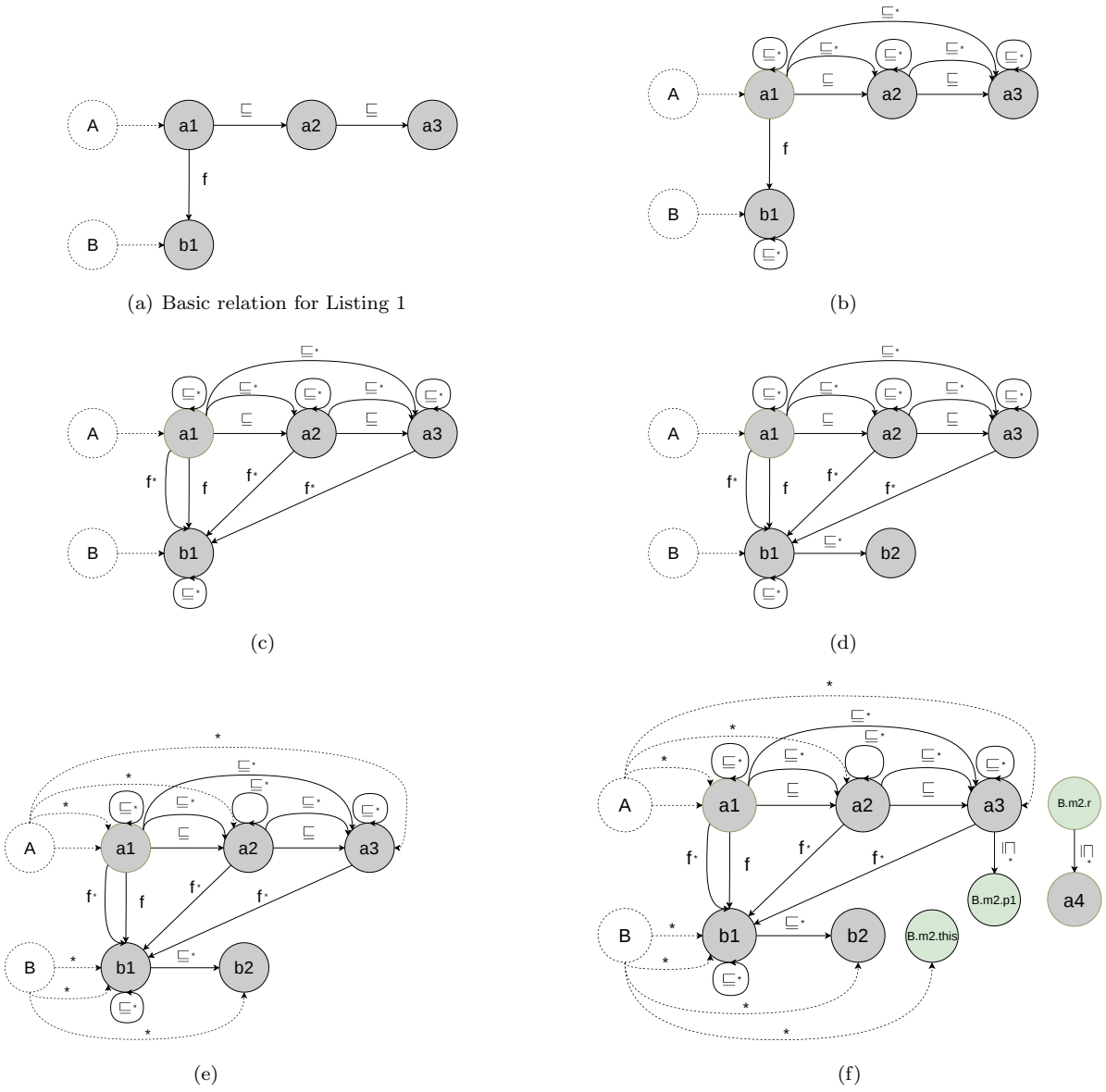
Intuitively, $c \dashrightarrow x$ means variable x may have type c (i.e., c flows to x), $y \sqsubseteq x$ means all types flow to y also flow to x, and $x \xrightarrow{f} y$ means that one may access variable y from field f together with variable x.[2] We give a snippet code, as shown in Listing 1, to illustrate the generation of relations. For statement "$A\ a1 = new\ A()$" and "$B\ b1 = new\ B()$", we generates two type flow relations, "$A \dashrightarrow a1$" and "$B \dashrightarrow b1$", respectively. Tow partial order relations, "$a1 \sqsubseteq a2$" and "$a2 \sqsubseteq a3$", are generated for the statements "$a2 = a1$" and "$a3 = a2$". For the field store statement on line 6. We generate a field access relation "$a1 \xrightarrow{f} b1$". The final result is shown is Figure. 6(a), on which a dash circle refers to type and other nodes in gray background refer to variables. Type relation, partial order relation and field access relation are represented in dash arrow, arrow with label "$\sqsubseteq$" and arrow with label "f", respectively.

```
1  public void m() {
2      A a1 = new A();
3      a2 = a1;
4      a3 = a2;
5      B b1 = new B();
6      a1.f = b1;
7      b2 = a1.f;
8      a4 = b2.m2(a3);
9  }
10
11 class B {
12     public A m2(p1) {... return r;}
13 }
```

Listing 1: Example for generation of relations

We use five rules to extend the basic relations for calculating a program fixpoint, which are given in the following definition 2. These rules intuitively represent propagation of type information. We put a '*' label behind relation symbol to destinguish from basic relations.

[2] Note that VTA treats statement $x.f = y$ as follows. For each class c that flows to x which defines field f, VTA assigns all types that flow to y also to $c.f$.



(a) Basic relation for Listing 1 (b)

(c) (d)

(e) (f)

Fig. 6. One round of relation generation for method $m()$ in Listing 1

Definition 2. (Extended Relations)

1. $y \sqsubseteq^* x$ if $x = y$ or $y \sqsubseteq x$ or $\exists z \in \texttt{VAR} : y \sqsubseteq^* z \wedge z \sqsubseteq^* x$;
   This rule is based on the properties of reflexive and transtive since $\sqsubseteq$ is a partial order relation. Besides, $y \sqsubseteq x$ implies $y \sqsubseteq^* x$. Under this rule, we extend the basic relations in 6(a) and show the result in 6(b).
2. $y \xrightarrow{f\ *} z$ if $\exists x \in \texttt{VAR} : x \xrightarrow{f} z \wedge (\exists z' \in \texttt{VAR} : z' \sqsubseteq^* y \wedge z' \sqsubseteq^* x)$;
   This rule is designed to extending field access relation. Based on the relations we generated in 6(b), we can generated three $\xrightarrow{f\ *}$ relations, "$a1 \xrightarrow{f\ *} b1$", "$a2 \xrightarrow{f\ *} b1$" and "$a3 \xrightarrow{f\ *} b1$", as displayed in 6(c).
3. For all statements $x = y.f$, if $y \xrightarrow{f\ *} z$, then $z \sqsubseteq^* x$.
   For field load statements, only we have generated a relavant field access relation, can we extend $\sqsubseteq^*$

statement	**Points-to constraints**
$x = \mathsf{new}\ c$	$o_i \in \Omega(x)$
$x = y$	$\Omega(y) \subseteq \Omega(x)$
$x = y.f$	$\forall o \in \Omega(y) : \Phi(o, f) \subseteq \Omega(x)$
$x.f = y$	$\forall o \in \Omega(x) : \Omega(y) \subseteq \Phi(o, f)$
$x = y.m(z)$	$\forall o \in \Omega(y) : \begin{cases} \Omega(z) \subseteq \Omega(param(type(o), m)) \\ \Omega(this(type(o), m)) = \{o\} \\ \forall x' \in return(type(o), m) : \\ \qquad \Omega(x') \subseteq \Omega(x) \end{cases}$

Fig. 7. Constraints for points-to analysis.

relation for target variables. *e.g.* , From the result of 6(c), we have a field access relation "$a1 \xrightarrow{f\ *} b1$". So we can generate a relation "$b1 \sqsubseteq^* b2$" for the field load statement "$b2 = a1.f$" on line 7 of Listing 1. We put the result on Fig. 6(d).

4. $c \dashrightarrow^* y$ if $c \dashrightarrow y$, or $\exists x \in \mathtt{VAR} : c \dashrightarrow^* x \wedge x \sqsubseteq^* y$;
   Intuitively, this rule describe how the type information being propagated over the $\sqsubseteq^*$ relation. We use this rule to extend the relations from 6(d). The generated relations are represented in dash arrow with a label of '*', as displayed in 6(e).
5. The type information is used to resolve each method call $x = y.m(z)$.

$$\begin{array}{l} \forall\ c \dashrightarrow^* y : \\ \forall\ m(z')\{\ldots \mathsf{return}\ x'\} \in methods(c) : \end{array} \begin{cases} z \sqsubseteq^* c.m.z' \\ c \dashrightarrow^* c.m.this \\ c.m.x' \sqsubseteq^* x \end{cases}$$

These three kinds of generated relation reprensent the type flow on parameter variables, "*this*" reference variable and return variables, respectively. The reaching type of variable $b2$ is $\{B\}$, *i.e.* , we have "$B \dashrightarrow^* b2$", which we can extract from the result of 6(e). That means statement "$a4 = b2.m2(a3)$" will trigger the invocation of method $m2()$ in class B. In this case, we generated three relations, "$a3 \sqsubseteq^* B.m2.p1$", "$B \dashrightarrow^* B.m2.this$" and "$B.m2.r \sqsubseteq^* a4$". The result is shown is 6(f), where variables in method $B.m2()$ is represented as circle in green background.

We show the detail process of relation generation in one round, as displayed in 6. In general we need more round to calculte a program fixpoint. The final relations are the least relations that satisfy constraints of Definition 2. Although the three relations are inter-dependent, one may find that without method call (i.e., Definition 2.5), a smallest model satisfying the two relations $\rightarrow^*$ (field access) and $\sqsubseteq^*$ (variable partial order) can be uniquely determined without considering the type flow relation $\dashrightarrow^*$.

In order to compare the precision of TFA with points-to analysis, we present a brief list of the classical subset-based points-to rules for our language in Figure 7, in which Ω (the var-points-to relation) maps a reference to a set of objects it may points to, and Φ (the heap-points-to relation) maps an object and a field to a set of objects. The points-to rules are mostly straightforward, except that $param(type(o), m))$, $this(type(o), m))$ and $return(type(o), m)$ refer to the formal parameter, this reference and return variable of method m of the class for which object o is declared, respectively.

To this end we present the first result of the paper, which says type flow analysis has the same precision regarding type based check, such as call site resolution and cast failure check, when comparing with the points-to analysis.

Theorem 1. In a context-insensitive setting, for all variables x and classes c, $c \dashrightarrow^* x$ in TFA iff there exists an object abstraction o of c such that $o \in \Omega(x)$ in points-to analysis.

Proof. (sketch) For a proof sketch, first we assume every object creation site $x = \mathsf{new}\ c_i$ at line i defines a mini-type c_i, and if the theorem is satisfied in this setting, a subsequent merging of mini-types into classes c will preserve the result.

Moreover, we only need to prove the intraprocedural setting which is the result of Lemma 1. Because if

in the intraprocedural setting the two systems have the same smallest model for all methods, then at each call site $x = y.m(a)$ both analyses will assign y the same set of classes and thus resolve the call site to the same set of method definitions, and as a consequence, each method body will be given the same set of extra conditions, thus all methods will have the same initial condition for the next round iteration. Therefore, both inter-procedural systems will eventually stabilize at the same model. □

The following lemma focuses on the key part of the proof for Theorem 1, which shows that TFA and points-to analysis are equivalent regarding call site resolution locally within a function.

Lemma 1. In a context-insensitive intraprocedural analysis where each class c only syntactically appears once in the form of $\texttt{new}\ c$, for all variables x and classes c, $c \dashrightarrow^* x$ iff there exists an object abstraction o of type c such that $o \in \Omega(x)$.

Proof. Since those constrains of the points-to analysis establish the smallest model (Ω, Φ) with $\Omega : \texttt{VAR} \to \texttt{OBJ}$ and $\Phi : \texttt{OBJ} \times \mathcal{F} \to \mathcal{P}(\texttt{OBJ})$, and the three relations of type flow analysis also define the smallest model that satisfies Definition 1 and Definition 2, we prove that every model established by points-to analysis constraints is also a model of TFA, and vice versa. Then the least model of both systems must be the same, as otherwise it would lead to contradiction.

($\star$) For the 'only if' part ($\Rightarrow$), we define a function $\mathit{Reaches}(x) = \{c \mid c \dashrightarrow^* x\}$ which maps a variable to its reaching types in TFA, and assume a bijection $\xi : \mathcal{C} \to \texttt{OBJ}$ that maps each class c to the unique (abstract) object o that is defined (and $\mathit{type}(o) = c$). Then we construct a function $\mathit{Access} : \mathcal{C} \times \mathcal{F} \to \mathcal{P}(\mathcal{C})$ and show that $(\xi(\mathit{Reaches}), \xi(\mathit{Access}))$ satisfies the points-to constraints. Define $\mathit{Access}(c, f) = \{c' \mid x \xrightarrow{f\ *} y \wedge c \in \mathit{Reaches}(x) \wedge c' \in \mathit{Reaches}(y)\}$. We prove the following cases according to the top four points-to constraints in Figure 7.

- For each statement $x = \texttt{new}\ c$, we have $\xi(c) \in \xi(\mathit{Reaches}(x))$;
- For each statement $x = y$, we have $\mathit{Reaches}(y) \subseteq \mathit{Reaches}(x)$ and $\xi(\mathit{Reaches}(y)) \subseteq \xi(\mathit{Reaches}(x))$;
- For each statement $x.f = y$, we have $x \xrightarrow{f} y$, then by definition for all $c \in \mathit{Reaches}(x)$, and $c' \in \mathit{Reaches}(y)$, we have $c' \in \mathit{Access}(c, f)$, therefore $\xi(c') \in \xi(\mathit{Reaches}(y))$ we have $\xi(c') \in \xi(\mathit{Access}(\xi(c), f))$.
- For each statement $x = y.f$, let $c \in \mathit{Reaches}(y)$, we need to show $\xi(\mathit{Access}(c, f)) \subseteq \xi(\mathit{Reaches}(x))$, or equivalently, $\mathit{Access}(c, f) \subseteq \mathit{Access}(x)$. Let $c' \in \mathit{Access}(c, f)$, then by definition, there exist z, z' such that $c \in \mathit{Reaches}(z)$, $c' \in \mathit{Reaches}(z')$ and $z \xrightarrow{f\ *} z'$. By $c \in \mathit{Reaches}(y)$ and Definition 2.4, we have $y \xrightarrow{f\ *} z'$. Then by Definition 2.1, $z' \sqsubseteq^* x$. Therefore $c' \in \mathit{Reaches}(x)$.

($\star$) For the 'if' part ($\Leftarrow$), let (Ω, Φ) be a model that satisfies all the top four constraints defined in Figure 7, and a bijection $\xi : \mathcal{C} \to \texttt{OBJ}$, we show the following constructed relations satisfy value points-to.

- For all types c and variables x, $c \dashrightarrow^* x$ if $\xi(c) \in \Omega(x)$;
- For all variables x and y, $x \sqsubseteq^* y$ if $\Omega(x) \subseteq \Omega(y)$;
- For all variables x and y, and for all fields f, $x \xrightarrow{f\ *} y$ if for all $o_1, o_2 \in \texttt{OBJ}$ such that $o_1 \in \Omega(x)$ and $o_2 \in \Omega(y)$ then $o_2 \in \Phi(o_1, f)$.

We check the following cases for the three relations $\dashrightarrow^*$, $\sqsubseteq^*$ and $\xrightarrow{*}$ that are just defined from the above.

- For each statement $x = \texttt{new}\ c$, we have $\xi(c) \in \Omega(x)$, so $c \dashrightarrow^* x$ by definition.
- For each statement $x = y$, we have $\Omega(y) \subseteq \Omega(x)$, therefore $y \sqsubseteq^* x$ by definition.
- For each statement $x.f = y$, we have for all $o_1 \in \Omega(x)$ and $o_2 \in \Omega(y)$, $o_2 \in \Phi(o_1, f)$, which derives $x \xrightarrow{f\ *} y$ by definition.
- For each statement $x = y.f$, given $y \xrightarrow{f\ *} z$, we need to show $z \sqsubseteq^* x$. Equivalently, by definition we have for all $o_1 \in \Omega(y)$ and $o_2 \in \Omega(z)$, $o_2 \in \Phi(o_1, f)$. Since points-to relation gives $\Phi(o_1, f) \subseteq \Omega(x)$, we have $o_2 \in \Omega(x)$, which derives $\Omega(z) \subseteq \Omega(x)$, the definition of $z \sqsubseteq^* x$.
- The proof for the properties in the rest of Definition 2 are related to transitivity of the three TFA relations, which are straightforward. We leave them for interested readers.

□

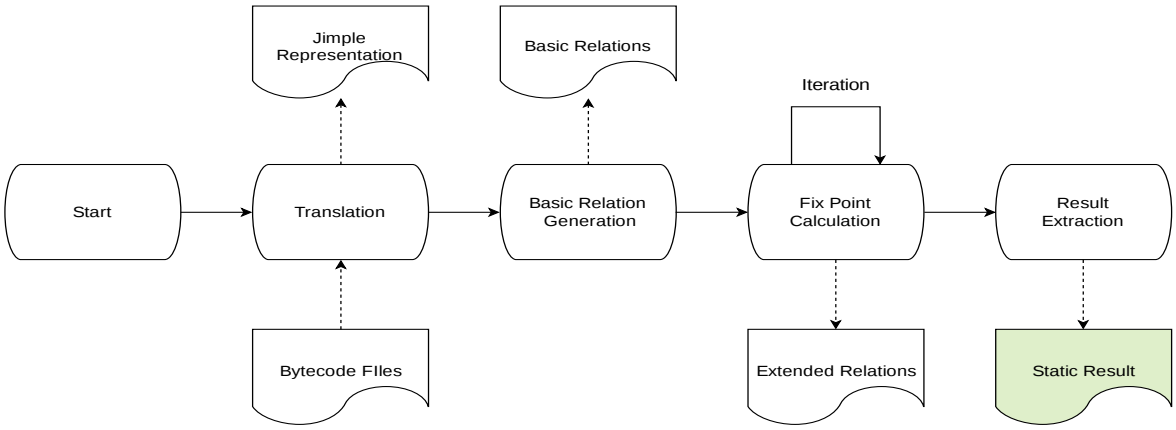


Fig. 8. Process of static analysis tool

3. Implementation

The analysis algorithm is written in Java, and is implemented in the Soot framework, the most popular static analysis framework for Java. We use jimple as our intermediate representation. We do not take common types (*e.g.* , int and float) under our consideration. That are irrelevant to our analysis. We keep method invocation from Java advanced features liked reflection or JNI as unresolvable. More detail will be discussed in 4.2.1 and 4.2.2. Conservative approximation is performed on invocation of methods from libraries (*e.g.* , JDK) and array accesses. We will describe these strategies in 4.2.3 and 4.2.4.

A Static analysis tool is implemented to process our algorithm and extract static result of type solution. In addition, we implement a dynamic profiler to record the run time type of variable, which can be used to compare with the static result. Detail of static analysis tool and dynamic profiler will be discuss in 3.1 and 3.2, respectively.

3.1. Static Analysis Tool

Our static analysis tool is implemented in Java and aims to analyze Java programs. It takes Java bytecode files as input. Any other format of Java code will be accepted if it can be translated to jimple representation by Soot(*e.g.* , jar files). The implementation of our static analysis tool can be splited into four step as follow.

- Code translation
  Target code is loaded by Soot and translated to IR of jimple format.
- Basic relation generation
  We iterate all statements on jimple IR to generate those basic relations based on the basic relation definitions.
- Fixpoint calculation
  After basic relations are generated, we use the extended relation rule to perform fixpoint calculation.
- Result extraction
  When the fixpoint is achieved, all set of reaching types of all variable are immutable. We extract those set of reaching types as our final result.

Fig 8 shows the whole process of our static analysis tool. Data input and generated are represented in dash arrow. The final result is generated in the “Result Extraction” phase and represented in green background.

3.2. Dynamic Profiler

Since the benchmarks do not provide the grountruth of run-time type of method receiver, we implement a dynamic profiling tool to record types which a receiver can access at run-time. To achieve this, we instrument statements into the target benchmark. After this instrumentation, the run-time type will be extracted during the benchmark execution. We consider this output as groundtruth and compare it with our static result in section 4.2. There are four manners to instrument the source code to record the run-time type of a method receiver.

- Insert First
  In this manner, the type-recorded statements will be insert before the first statement of a method block and the type of "this" reference in that method will be recorded. The reason we only have to record "this" reference is that a receiver is always passed into "this" reference in a method, except for static methods.

- Insert Before
  Statements will be inserted before invocations and the type of receiver will be recorded in this manner. It is more straightforward than the method we discuss about recording "this" reference.

- Insert Last
  This manner is similar with "Insert First", except that statements are inserted after the last statement of a method block. We also record "this" reference in this way.

- Insert After
  Statments will be inserted right after invocations and the type of receiver will be recorded. It is similar with "Insert Before", except that statements are inserted at different position. We take this manner in our implementation and the reason will be discuss in section 3.2.1

3.2.1. Our Instrumentation Manner

We take "Insert After" as our instrumentation manner. The reason is mainly due to the Java specification of constructor that the first statement in constructor should be either another constructor of its own or its super class. Therefore, we will get JVM voilation error if we instrument a statement before the first statement in construtor. We illustrate this example on Listing 2. So both "Insert First" and "Insert Before" manners can not be applied under this circumstance. We choose "Insert After" over "Insert Last" for reason that it's more straightforward. The code before and after instrumentation are shown in Listing 3 and Listing 4, respectively. For Listing 4, at line 5, the funtion $RecordUtils.id()$ takes an invocation expression ($invokeExprssion$) and a method receiver (b) as parameters and return an unique representation string for this invocation. For this example, the unique representation is "A:m1:b:B:m2:4". This means b will call method $m2$ of class B, at line 4 in method $m1$ of class A. It also shows that b is of type B at this position.

```
1  class A {
2    public A() {
3      //insert statements here will violate JVM specification
4      super();    //invoke super class constructor
5    }
6    public A(int i) {
7      //insert statements here will violate JVM specification
8      this();    //inovke another constructor of its own
9    }
10 }
```

Listing 2: Java specification on constructor

```
1 class A {
2   public void m1() {
3     B b = new B();
4     b.m2();     //invocation here
5   }
6 }
```

Listing 3: Example code before instrumentation

```
1 class A {
2   public void m1() {
3     B b = new B();
4     b.m2();     //invocation here
5     String record = RecordUtils.id(invokeExpression, b);
6     RecordUtils.record(record);
7   }
8 }
```

Listing 4: Example code after instrumentation

4. Evaluation

We evaluate our approach by measuring its performance on SPECjvm2008, which contains 12 benchmark programs in total. We conduct all of our experiments on a laptop equipped with an Intel i5-8250U CPU at 1.60 GHz and 8 GB memory, running Ubuntu 16.04LTS with OpenJDK 1.8.0.

We compare our approach against the default implementation of Class Hierarchy Analysis (CHA) and context-insensitive points-to analysis that are implemented by Soot team. The reason we do not compare against Variable Type Analysis (VTA) due to its unavaliable implementation. The only avaliable implementation for Java is also implemented by Soot team, but it is embeded as a subprocess to optimize points-to analysis. Under this circumstance, we compare our approach against VTA in precision with manual analysis in section 1. Implementation of VTA will be left for our future work. The choice of the context-insensitive points-to analysis is due to our approach also being context-insensitive, thus the results will be comparable. We use iteration algorithm for points-to analysis to calculate a fixpoint for the same reason. In the following tables we use CHA, PTA and TFA to refer to the results related to class hierarchy analysis, points-to analysis and type flow analysis, respectively. During the evaluation the following tow research questions are addressed.

- **RQ1** How efficient is our approach compared with the traditional class hierarchy analysis and points-to analysis?
- **RQ2** How accurate is our algorithm when comparing with the other analysis?

We answer these two questions in section 4.1 and section 4.2, respetively.

4.1. Efficiency

We executed each benchmark program 10 times with the CHA, PTA and TFA algorithms. We calculated the average time consumption (in seconds) as displayed at column $\mathbf{T}_{CHA}$(s), $\mathbf{T}_{PTA}$(s) and $\mathbf{T}_{TFA}$(s) of the Table in Fig. 9. We counted the sizes of each generated realtion (*i.e.* , the type flow realtion '$\dashrightarrow$', variable partial order '$\sqsubseteq$' and the field access relation '$\longrightarrow$'). It provides an estimation of size for the problem we are treating. The result shows that our approach consums more time than CHA, for the reason that CHA do not have to analyze detail logic inside the program but only analyze the class and interface hierarchical structure. TFA is in general more efficient than points-to analysis. The runtime cost in TFA basically depends on the size of generated relations, as well as the relational complexity as most of the time is consumed to calculate a fixpoint. For PTA it also require extra time for maintaining and updating a heap abstraction.

Benchmark	$\mathbf{T_{CHA}}(s)$	$\mathbf{T_{PTA}}(s)$	$\mathbf{T_{TFA}}(s)$	$\mathbf{R}\dashrightarrow$	$\mathbf{R}_{\sqsubseteq}$	$\mathbf{R}\rightarrow$	$\mathbf{R}_{total}$
check	5.33	71.98	10.46	6847	15599	10089	32535
compiler	5.76	72.01	11.48	6665	14898	10433	31996
compress	5.72	71.06	12.19	6410	14344	10322	31076
crypto	5.60	69.25	9.21	6459	14424	10152	31035
derby	5.83	70.57	11.65	6887	14853	10603	32343
helloworld	6.26	70.72	13.10	6149	13652	10071	29872
mpegaudio	6.29	70.45	10.65	6197	13737	10084	30018
scimark	6.77	71.37	11.24	6366	14678	10214	31258
serial	7.06	70.41	10.98	6627	14309	10341	31277
startup	5.62	69.09	10.73	6239	13723	10094	30056
sunflow	5.58	72.93	11.12	6167	13675	10077	29919
xml	6.55	139.08	13.84	6866	16039	11060	33965

Fig. 9. Runtime cost with different analysis

[1] $\mathbf{R}_{relation}$ denotes different type of relation we generated.

4.2. Accuracy

We answer the seconde question about accuracy in two indexes: precision and recall. Their definitions are given in equation 1 and 2, respectively.

$$precision = \frac{M}{S} \tag{1}$$

$$recall = \frac{M}{D} \tag{2}$$

"D" stands for the number of dynamic record, "S" refers to the numer of static record, and "M" represents the number of matching record.

A sound algorithm should catch all type information in run time, *i.e.* , the recall should be 100%. Precision indicates how accurate the generated set of types is. In general, a more acurrate algorithm often generate a smaller set of types for each calling variable. We calculate recall and precision of different approach, as displayed in Fig. 10

The result shows that in general our approach can achieve higher precision than CHA and closer precision than PTA. The imprecision is mainly due to our approximation on library invocation and array, which we will discuss in section 4.2.3 and section 4.2.4. Besides, the code coverage will issue imprecision since SPECjvm provides common codes for all benchmarks and some of them would not be excuted when a specific benchmark is excuting, *i.e.* , these codes will be analyzed statically but not generate any record in run time.

For soundness, the recall of both TFA and PTA can not achieve to 1.00 in most benchmarks except *helloworld*. We analyzed those missed records and find out the reason is due to some Java advance features, liked reflection call and JNI call, which we will describe in section 4.2.1 and section 4.2.2. In addition, callback mechanism will raise unsoundness under our treatment on reflection, JNI and library call. We will discuss it in section 4.2.5. The result shows that PTA produces lower recall than TFA in general. We study the reason for this gap from read the source code of Soot and find out it is due to the on-the-fly reachability analysis used on PTA implemented by Soot team. Method invoked by reflection, or invoked as a callback function inside reflection, JNI and library, will be denoted as unreachable under this analysis. As a consequence, some methods would not be analyzed by PTA. Note that even the unreachable method would not be analyzed, a variable receiving its type from JDK would be assigned to possible types based on the preprocessing from Soot. Such that the set of possible reaching types for these variables would not be empty.

4.2.1. Reflection Call

Reflection in Java programming language is a advanced feature which provides ability to inspect and manipulate a Java class at runtime. It brings in extra complexity on programs and the behaviour is hard to

Benchmark	**D**	$\mathbf{M}_{CHA}$	$\mathbf{M}_{PTA}$	$\mathbf{M}_{TFA}$	$\mathbf{R}_{CHA}$	$\mathbf{R}_{PTA}$	$\mathbf{R}_{TFA}$	$\mathbf{P}_{CHA}$	$\mathbf{P}_{PTA}$	$\mathbf{P}_{TFA}$
check	150	150	145	145	1.00	0.97	0.97	0.07	0.19	0.18
compiler	515	515	463	506	1.00	0.90	0.98	0.28	0.81	0.70
compress	353	353	330	345	1.00	0.93	0.98	0.19	0.49	0.46
crypto	454	454	393	407	1.00	0.87	0.90	0.25	0.70	0.65
derby	639	639	603	632	1.00	0.94	0.99	0.35	0.97	0.88
helloworld	21	21	21	21	1.00	1.00	1.00	0.01	0.03	0.03
mpegaudio	255	255	241	248	1.00	0.95	0.97	0.15	0.40	0.37
scimark	386	386	357	362	1.00	0.92	0.94	0.20	0.54	0.50
serial	458	458	354	410	1.00	0.77	0.90	0.18	0.67	0.62
startup	1429	1429	1358	1360	1.00	0.95	0.95	0.82	2.23	2.07
sunflow	226	226	208	217	1.00	0.92	0.96	0.13	0.36	0.33
xml	521	521	436	504	1.00	0.84	0.97	0.27	0.81	0.65

Fig. 10. Accuracy with different analysis

[1] **D** denotes dynamic records.
[2] $\mathbf{M}_{algorithm}$, $\mathbf{R}_{algorithm}$, $\mathbf{P}_{algorithm}$ denote matching count, recall and precision of different algorithms, respectively.

predict statically. In Listing 5 we pick some codes using reflection in the benchmark programs to discuss how reflection works and what is our treatment on that. Note that we reorganize and simplify the real code a little to concentrate on the main point of reflection usage. We discuss in three cases:

- Object Creation
  Related codes range from line 6 to 10. A new obejct of type "SPECJVMBenchmarkBase" will be created by invoking the method "$newInstance()$" on variable c at line 10. c is an object of "Constructor" type and it refers to a specific constructor of class "SPECJVMBenchmarkBase". Our method discard the type information of new object in these case because it's difficult to statically identify which constructor will be invoked. *e.g.* , At line 6, If statement $Class.forName()$ receive argument from outside liked user input or loading file with content of class name, then we could not find out the real type of $benchmarkClass$. As a result, the run time type of c and $benchmark$ could not be identified neither.

- Method Invocation
  After a new object is instantiated, we can get an object of "Method" type, referring to a specific method of a class, and call the method named "invoke()" of that object. This effect is just like a normal invocation at line 12. The invocation receiver is passed to the first argument of method "invoke()". If this invocation is static, a **null** reference will be passed to the first argument. We do not consider these effects of method invocation in reflection manner for the same reason we discuss about object creation.

- Field Modification
  The way to change a field using reflection is similar to processing method invocation. The last two line illustrate changing value of a field named "f" on object "benchmark" into a new object. We discard this effect as well because of the difficulty on analyzing which class holds the target field.

```
1  public static void runSimple(Class benchmarkClass, String [] args) {
2    ...
3    ...
4    Class[] cArgs = { BenchmarkResult.class, int.class };
5    Object[] inArgs = { bmResult, Integer.valueOf(1)};
6    Class benchmarkClass = Class.forName("spec.harness.SpecJVMBenchmarkBase");
7    Constructor c = benchmarkClass.getConstructor(cArgs);
8
9    // Object creation using reflection
10   SpecJVMBenchmarkBase benchmark = (SpecJVMBenchmarkBase)c.newInstance(inArgs);
11   // normal method invocation
12   benchmark.harnessMain();
13
14   // method invocation
15   Method harnessMain = benchmarkClass.getMethod("harnessMain", new Class[]{});
16   // just like line 11 but in reflection manner
17   harnessMain.invoke(benchmark, new Object[]{});
18
19   Method setup = benchmarkClass.getMethod( "setupBenchmark", new Class[]{});
20   // static invocation
21   setup.invoke(null, new Object[]{});
22
23   // field modification
24   Field f = benchmarkClass.getField("f");
25   f.set(benchmark, new Object());
26 }
```

Listing 5: Example code of reflection

4.2.2. Java Native Interface Call

Java Native Interface (JNI) is a standard Java programming interface which provide ability for Java code to interoperate with application or library written in other programming languages, such as C, C++ or assembly. We show the usage of JNI in Listing 6. Method "$m()$" is defined as a native method and should not be implemented in Java. This program will load a native library named "*lib*", in which the method "$m()$" is actually implemented in different program languages. We do not consider JNI calls in our algorithm since the code is not written in Java. Analyzing that code and the communication between Java and other languages are out of our research scope. As a consequence, the effect of that invocation "$a.m()$" at line 8 will be discarded.

```
1  public class A {
2    public native void m();
3    static {
4      System.loadLibrary("lib");
5    }
6    public static void main(String[] args) {
7      A a = new A();
8      a.m(); <---
9    }
10 }
```

Listing 6: Example code of JNI

4.2.3. Library

Library are those codes included in the application and used to accomplish specific function(*e.g.* , JDK library, three-party library). Listing 7 shows a common case of JDK invocation. We do not analyze the detail logic inside library code which are written at line 10-11. Instead, we perform an over approximation on library invocation, based on the method definition which appears at line 9. We assume that library invocation will return the definition type and any subtype of this definition type as the result type. For Listing 7, *sb*2 will receive {*StringBuilder*, *any_subtype_of_StringBuilder*} as the set of reaching types under this over approximation strategy.

```
1  import java.lang.StringBuilder;
2
3  public void m() {
4    StringBuilder sb = new StringBuilder();
5    StringBuilder sb2 = sb.append("abc"); <---
6  }
7
8  //  @Override
9  //  public StringBuilder append(String str) {
10 //    ...
11 //    ...
12 //  }
```

Listing 7: Example of JDK library call

4.2.4. Array Approximation

We perform a conservative treatment on array accesses that all type information that flows to one member of an array flows to all members of that array. Codes in Listing 8 are used to explain how this approximation works. Type information of A and B are flow to the first member and the second member of array *arr*, respectively. We approximate that these two type information flow to array *arr*. Loadding an element of array will receive all types that array can access(*e.g.* , b will receive $\{A, B\}$ as the set of reaching types, which is the same set of types that array *arr* can access).

```
1  public void m() {
2      Object[] arr = new Object[2]{};
3      arr[1] = new A();
4      arr[2] = new B();
5      Object b = arr[1];  <---
6  }
```

Listing 8: Example of array access

4.2.5. Callback

A callback function is known as a "call-after" function. It is widely used in programs. By passing a argument to one invocation as the receiver which is expected to trigger a callback invocation. In functional language a callback funtion can be passed as argument directly. Listing 9 shows the callback mechanism. The function *callback*() will be invoked after the invocation of *trigger*(). Since we do not analyze the effect of reflection call, JNI call, and library call, we are unable to catch the type flow information inside *a.trigger*(*b*) if it happens to be one of these calls (*e.g.* , *trigger*() is a method of a library class). Note that we only conservatively approximate on the return type of a library invocation, but not analyze the detail logic inside a library method.

```
1  class A {
2      public void trigger(B b) {
3          b.callback();  // here the callback function actually executes
4      }
5  }
6
7  class B {
8      public void callback() {...}
9      public void static main() {
10         A a = new A();
11         B b = new B();
12         a.trigger(b);
13     }
14 }
```

Listing 9: Callback mechanism

5. Related Work

There are not many works focusing on general purpose call graph construction algorithms, and we give a brief review of these works first.

As stated in the introduction, Class Hierarchy Analysis (CHA) [**?**, **?**], Rapid Type Analysis (RTA) [**?**] and Variable Type Analysis (VTA) [**?**] are efficient algorithms that conservatively resolves call sites without any help from points-to analysis. Grove et al. [**?**] introduced an approach to model context-sensitive and context-insensitive call graph construction. They define call graph in terms of three algorithm-specific parameter partial orders, and provide a method called Monotonic Refinement, potentially adding to the class sets of local variables and adding new contours to call sites, load sites, and store sites. Tip and Palsberg [**?**] Proposed four propagation-based call graph construction algorithms, CTA, MTA, FTA and XTA. CTA uses distinct sets for classes, MTA uses distinct sets for classes and fields, FTA uses distinct sets for classes and methods, and XTA uses distinct sets for classes, fields, and methods. The constructed call graphs tend to contain slightly fewer method definitions when compared to RTA. It has been shown that associating a distinct set of types with each method in a class has a significantly greater impact on precision than using a distinct set for each field in a class. Reif et al. [**?**] study the construction of call graphs for Java libraries that abstract over all potential library usages, in a so-called *open world* approach. They invented two concrete call graph algorithms for libraries based on adaptations of the CHA algorithm, to be used for software quality and security issues. In general they are interested in analyzing library without knowing client application, which is complementary to our work that has focus on client program while treating library calls as end nodes.

Call graphs may serve as a basis for points-to analysis, but often a points-to analysis implicitly computes a call graph on-the-fly, such as the context insensitive points-to algorithm implemented in Soot using SPARK [**?**]. Most context-sensitive points-to analysis algorithms (e.g., [**?**, **?**, **?**, **?**]) progress call edges together with value flow, to our knowledge. The main distinction of our approach from these points-to analysis is the usage of an abstract heap, as we are only interested in the actual reaching types of the receiver of a call. Nevertheless, unlike CHA and VTA, our methodology can be extended to context-sensitive settings.

Regarding the treatment of flow analysis in our algorithm, downcast analysis has been studied in region inference which is a special memory management scheme for preventing dangling pointers or improving precision in garbage collection in object-oriented programming languages [**?**, **?**]. These works are type-based analysis, while our methodology belongs to traditional static program analysis. Similar ideas regarding value flow can also be found in the graph-reachability based formulation (e.g. [**?**, **?**]) to which all distributed data flow analyses can be adopted.

Runtime profiling data can be recorded and used to compare with static analysis result. Furthermore, these runtime data can help static algorithms to increase precision and achieve adaptive ability. In general there are two ways to record runtime profiling data for Java programs, collecting from Java VM [**?**], or using program instrumentation technique to extract these data in runtime [**?**]. Inspire by [**?**], and partialy due to

Java VM can only internally provide the data we want, our dynamic type recorder is implemented using instumented technique.

6. Conclusion and future work

In this paper we have proposed Type Flow Analysis (TFA), an algorithm that statically determines reaching types of variable and constructs call graph edges for Object-Oriented programming languages. We have proved that our method is as precise as subset-based points-to analysis, regarding type related information. The computation of TFA is pure relational since it do not require a heap abstraction, which is needed by points-to analysis.

To evaluate our method in practice, we implement a static type analysis tool in Soot framework, which runs our algorithm. Moreover, a dynamic type profiler is used to extract run-time types of variables, which are then used to compare with static results. We implement it using instrumentation technique, abiding by specification of Java VM. We have compared our method with built-in CHA and points-to algorithms in Soot by experimental evaluation. The result have shown that our method achieved promising result. It is generally more practical in type resultion problem.

There are still some works we can do to enhance our algorithm. On theoretical aspect, we can develop our algorithm to context-sensitive, in a way similar to points-to analysis [**?**, **?**]. On practical aspect, we can combine static result and dynamic result to make our algorithm more adaptive, inspire by the approach proposed in [**?**].

Acknowlegements