Under consideration for publication in Formal Aspects of Computing

TFA: An Efficient and Precise Virtual Method Call Resolution for Java

Xilong Zhuo[⋆] and Chenyi Zhang[⋆]

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

Abstract. The problem of statically resolving virtual method calls in object-oriented (OO) programming languages has been a long standing challenge, often due to the overly complicated class hierarchy structures in modern OO programming languages such as Java, C# and C++. Traditional ways of dealing with this problem include class hierarchy analysis (CHA), variable type analysis (VTA), and retrieval of type information after a sophisticated points-to analysis. In this paper, we tackle this problem by proposing a new approach called type flow analysis (TFA) which propagates type information as well as field access information through the syntactic structure of a program. Our methodology is purely algebraic and there is no need to explicitly construct a heap abstraction.

We have assessed our methodology from two perspectives. Regarding its theoretical foundation, we have proved that in the context insensitive setting, our method is as precise as the standard Andersen's subset-based points-to analysis regarding the derived types for variables. For an experimental evaluation of TFA, we have implemented the algorithm in the Soot framework and used it to analyze the SPECjvm2008 benchmark suite. During the experiment, we have shown that our method is usually $30 - 100$ times faster than the standard points-to analysis. We further conduct a range of detailed analysis based on the baseline data obtained by running a dynamic profiler, which is also implemented by us, on the SPECjvm2008. The experiment results confirm that TFA can achieve outstanding performance with acceptable accuracy when applied on real world Java programs.

Keywords: Type Analysis; Static Analysis; Virtual Method Resolution; Call Graph Construction

1. Introduction

For object-oriented programming languages, virtual methods (or functions) are those declared in a base class but are meant to be overridden independently in child classes. Statically determine a set of methods that may be invoked at a call site is of vital importance to program optimization. From results of a precise static method call resolution, a subsequent optimization process may be applied to reduce the cost of virtual

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

[1] A preliminary version appears in the proceedings of the 21st International Conference on Formal Engineering Methods (ICFEM 2019) held in Shenzhen, China, November 5th-9th, 2019.

```
class A{
    A f;
    A 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.

function calls, or to perform method inlining if target method forms a singleton set. From these results one may also remove methods that are never called by any call sites (*i.e.* dead code elimination), or produce a call graph which can be useful in other optimization processes.

Efficient solutions, such as Class Hierarchy Analysis (CHA) [?, ?] and Rapid Type Analysis (RTA) [?], conservatively assign each variable a set of class definitions, with relatively low precision. Variable Type Analysis (VTA) [?] collects types from the object instantiation sites and then propagates type information within the program. However, VTA does not differentiate variables of the same class, which potentially causes imprecision. Alternatively, with the help of a heap abstraction, one may take advantage of points-to analysis [?] to compute a set of objects that a variable may refer to, and resolve the receiver classes by retrieving the types of the objects, before determining associated methods at call sites.

In general, 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 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 (syntactically) 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 $1 - 4$ of Figure 1 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 undersirable extra information 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 $\mathtt{VAR}$, where $x \sqsubseteq y$ denotes all types that flow to x also flow to y, we also build a ternary field store relation $\rightarrow \subseteq \mathtt{VAR} \times \mathcal{F} \times \mathtt{VAR}$ 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

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

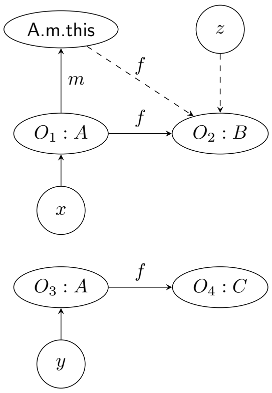


Fig. 3. Points-to results on the example

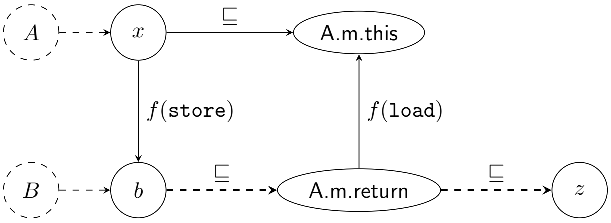


Fig. 4. Type Flow Analysis for variable z in the example

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 (which we refer to as PTA in the rest of the paper) 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 IR, a typed, 3-address, statement based intermediate representation provided by Soot framework [**?**]. The implementation has been tested on a collection of benchmark programs from SPECjvm2008 [**?**]. We conducted several experiments to compare TFA regarding efficiency and accuracy with CHA and PTA implemented by the Soot team. In order to establish a reliable comparison regarding *precision* and *recall* of different approaches, we apply the instrumentation technique to implement a dynamic profiling tool for jimple, which extracts types of callee method at runtime. We then conduct a sophisticated study on the different static approaches regarding results generated by the dynamic profiler. The experimental result has shown that TFA is in general more efficient than PTA. In terms of accuracy, TFA is more precise than CHA. Note that due to the comprehensive optimizations for VTA and PTA that are implemented by the Soot team, a decent comparison to the VTA and PTA implementation available in Soot is not straightforward. We have tried to create a relatively suitable environment by disabling some of the optimizations in Soot, and eventually reach the conclusion that the precision of TFA is comparable to VTA and PTA regarding the SPECjvm2008 benchmark suite. In addition, we confirm that most existing approaches, including PTA and TFA, do not guarantee soundness on all benchmarks, as already discussed in the literature [**?**]. We then look into all the unmatch record to find out the source of imprecision and unsoundness, with short discussions regarding the related Java language features such as JDK library calls, Java reflection API, Java Native Interface and the callback mechanism. Our implementation is publicly available at `https://github.com/SeanCoek/tfa`.

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

Fig. 5. Abstract syntax for the core language.

Over all the contributions of the paper are listed as follows.

1. We have proposed a new relational static semantics called Type Flow Analysis (TFA), which computes a collection of reachable types for variables. Our methodology is enforced by type propagation via value flow between program variables as well field accesses, following predefined relational rules (C.f. Definition 1 and Definition 2). We have proved equivalence between TFA and the subset based points-to analysis regarding collected types for variables (Theorem 1), presented in Section 2.
2. We have implemented the TFA algorithm in the Soot framework, and conducted a comprehensive study regarding performance and accuracy. The detailed description regarding the implementation and comparison with existing approaches (CHA, VTA and PTA) are presented in Section 3 and Section 4.
3. As a byproduct, we have implemented a dynamic profiler, by instrumenting code that collects at each call site a set of types of the receiver variable as well as callee methods. The collected type information serves as a baseline for measuring and comparing the precisions between TFA and the aforementioned virtual method call resolution algorithms.

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. [2]

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 method. 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 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 base relations represent program facts which are generated directly from statements syntactically appearing in programs.

1. $c \dashrightarrow x$ if there is a statement $x = \textsf{new}\ c$;

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

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 that 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.[3]

Example 1. We give a snippet code 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. We also generate "$a1 \sqsubseteq a2$" and "$a2 \sqsubseteq a3$" from 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 resulting base relations are shown in Figure. 6(a), where dashed circles refer to types and solid nodes in gray colour represent variables. The type relation, partial order relation and field access relation are shown in dashed arrows, solid arrows labelled "$\sqsubseteq$" and solid arrows labelled with field names.

```
 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: An example showing the generation of base relations

In the following we define extended relations to capture propagation of type information inter-procedurally in programs. As a convention, R^* (Kleene star) is used to represent the reflexive and transitive closure of relation R. Then we explain the associated operations in Example 2, illustrated in Figure 6.

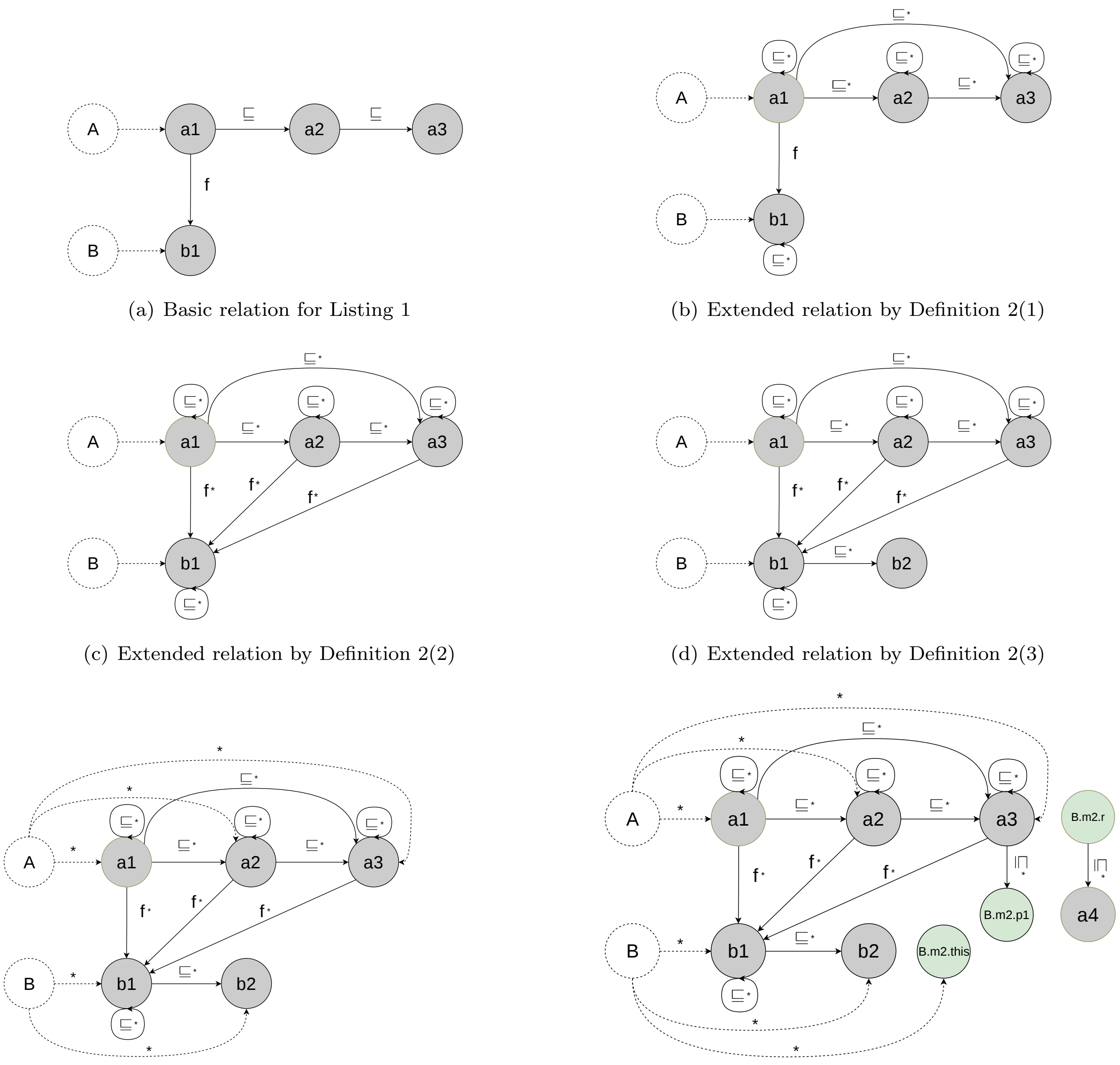
Definition 2. (Extended Relations)

1. $y \sqsubseteq^* x$ if $x = y$ or $y \sqsubseteq x$ or $\exists z \in \mathtt{VAR} : y \sqsubseteq^* z \wedge z \sqsubseteq^* x$;
2. $y \xrightarrow{f\ *} z$ if $\exists x \in \mathtt{VAR} : x \xrightarrow{f} z \wedge (\exists z' \in \mathtt{VAR} : z' \sqsubseteq^* y \wedge z' \sqsubseteq^* x)$;
3. For all statements $x = y.f$, if $y \xrightarrow{f\ *} z$, then $z \sqsubseteq^* x$.
4. $c \dashrightarrow^* y$ if $c \dashrightarrow y$, or $\exists x \in \mathtt{VAR} : c \dashrightarrow^* x \wedge x \sqsubseteq^* y$;
5. The type information is used to resolve each method call $x = y.m(z)$.

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

Regarding the extended relations given in Definition 2, we provide the following intuitions. Definition 2(1) computes the reflexive and transtive closure of the relation $\sqsubseteq$. In Definition 2(2), with the existence of z', variables x and y are in the *may-alias* relation, therefore, all that is accessible by x via field f is also accessible by y via f. This rule is designed to extend the field access relation. Definition 2(3) loads z to x (via f) from y, provided that z is accessible by y via f. Definition 2(4) extends the type flow relation by taking advantage of the extended partial order $\sqsubseteq^*$ on variables. Definition 2(5) deals with inter-procedural type flow, which defines three extensions for parameter variables, `this` references and `return` variables, respectively.

[3] Note that VTA treats statement $x.f = y$ in a different way 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) Extended relation by Definition 2(1)

(c) Extended relation by Definition 2(2)

(d) Extended relation by Definition 2(3)

(e) Extended relation by Definition 2(4)

(f) Extended relation by Definition 2(5)

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

Example 2. We describe the process of generating extended relations for the base model that is discussed in Example 1, as depicted in Figure 6. Although the actual computation may take a number of iterations to reach a fixpoint, to make the process more understandable, we try to sketch in separate how each rule given in Figure 6 works. Initially, the base relation for Listing 1 is given in Figure 6(a). Definition 2(1) extends the base relation $\sqsubseteq$ into $\sqsubseteq^*$, as shown in Figure 6(b). Definition 2(2) extends the field access relation, resulting in Figure 6(c). By applying Definition 2(3), we further extends $\sqsubseteq^*$ through the existing field access relation, reaching the graph in Figure 6(d). Then Definition 2(4) propagates the type flow relation $\dashrightarrow^*$ by considering the existing partial order relation and field access relation on variables, as shown in Figure 6(e). In Figure 6(f) we further enrich the relations for inter-procedural type flow, where the statement "$a4 = b2.m2(a3)$" triggers the invocation of method $m2()$ in class B, from which we establish "$a3 \sqsubseteq^* B.m2.p1$", "$B \dashrightarrow^* B.m2.this$" and "$B.m2.r \sqsubseteq^* a4$", where $B.m2.p1$ and $B.m2.r$ are respectively the virtual parameter of method $m2()$

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

Fig. 7. Constraints for an context-insensitive points-to analysis.

and the return variable of method $m2()$. In fact, Figure 6(f) is very close to the smallest type flow model satisfying all constraints given in Definition 2 regarding the program in Listing 1.

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 point 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 the 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) First we assume that for each class c, every object creation site $x = \textsf{new}\ c_i$ 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 class c will preserve the result.

Moreover, we only need to prove the result in the intraprocedural setting (*i.e.* 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 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 reachable types to variables intraprocedurally.

Lemma 1. In a context-insensitive intraprocedural analysis where each class c only syntactically appears once in the form of $\textsf{new}\ c$, 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 $\Omega : \mathtt{VAR} \rightarrow \mathtt{OBJ}$ and $\Phi : \mathtt{OBJ} \times \mathcal{F} \rightarrow \mathcal{P}(\mathtt{OBJ})$, 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 $Reaches(x) = \{c \mid c \dashrightarrow^* x\}$ which maps a variable to its reaching types in TFA, and assume a bijection $\xi : \mathcal{C} \rightarrow \mathtt{OBJ}$ that maps each class c to the unique (abstract) object o that is defined (and $type(o) = c$). Then we construct a function $Access : \mathcal{C} \times \mathcal{F} \rightarrow \mathcal{P}(\mathcal{C})$ and show that $(\xi(Reaches), \xi(Access))$ satisfies the points-to constraints. Define $Access(c, f) = \{c' \mid x \xrightarrow{f\ *}$

$y \wedge c \in \mathit{Reaches}(x) \wedge c' \in \mathit{Reaches}(y)\}$. We prove the following cases according to the top four points-to constraints in Figure 7.

- For each statement $x = \mathsf{new}\ c$, we have $\xi(c) \in \xi(\mathit{Reaches}(x))$, as $c \in \mathit{Reaches}(x)$ by Definition 1(1);
- For each statement $x = y$, we have $\mathit{Reaches}(y) \subseteq \mathit{Reaches}(x)$ and $\xi(\mathit{Reaches}(y)) \subseteq \xi(\mathit{Reaches}(x))$, as $y \sqsubseteq^* x$ by Definition 1(2);
- For each statement $x.f = y$, we have $x \xrightarrow{f} y$ by Definition 1(3), therefore $x \xrightarrow{f}{}^{*}\ y$. We need to show that for all $c \in \mathit{Reaches}(x)$, $\xi(\mathit{Reaches}(y)) \subseteq \xi(\mathit{Access}(c, f))$. Let $c \in \mathit{Reaches}(x)$, and $c' \in \mathit{Reaches}(y)$, by definition we have $c' \in \mathit{Access}(c, f)$. Therefore $\mathit{Reaches}(y) \subseteq \mathit{Access}(c, f)$. Then consequently, $\xi(\mathit{Reaches}(y)) \subseteq \xi(\mathit{Access}(c, f))$.
- For each statement $x = y.f$, for all $c \in \mathit{Reaches}(y)$, we need to show $\xi(\mathit{Access}(c, f)) \subseteq \xi(\mathit{Reaches}(x))$, or equivalently, $\mathit{Access}(c, f) \subseteq \mathit{Reaches}(x)$. Let $c' \in \mathit{Access}(c, f)$, then by definition of Access, there exist z, z' such that $c \in \mathit{Reaches}(z)$, $c' \in \mathit{Reaches}(z')$ and $z \xrightarrow{f\ *} z'$. Since each type is uniquely created in the current setting, *i.e.* there exists a statement $z'' = \mathsf{new}\ c$ such that $z'' \sqsubseteq^* y$ and $z'' \sqsubseteq^* z$. Then by $z \xrightarrow{f\ *} z'$ and Definition 2(2), we have $y \xrightarrow{f\ *} z'$. Then by $x = y.f$ and Definition 2(3), we have $z' \sqsubseteq^* x$. 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 $\xi : \mathcal{C} \to \mathtt{OBJ}$ between class names and objects, we show that the following constructed relations ($\dashrightarrow^*$, $\sqsubseteq^*$, $\xrightarrow{*}$) derived from (Ω, Φ) satisfy the (intraprocedural) constraints defined for the base relations (*i.e.* Definition 1 and the extended relations (*i.e.* Definition 2) for TFA.

- For all types c and variables x, $c \dashrightarrow^* x$ iff $\xi(c) \in \Omega(x)$;
- For all variables x and y, $x \sqsubseteq^* y$ iff $\Omega(x) \subseteq \Omega(y)$;
- For all variables x and y, and for all fields f,
  – $x \xrightarrow{f} y$ iff for all $o \in \mathtt{OBJ}$, $o \in \Omega(x)$ implies $\Omega(y) \subseteq \Phi(o, f)$.
  – $x \xrightarrow{f\ *} y$ iff there exists $o \in \Omega(x)$, $\Omega(y) \subseteq \Phi(o, f)$.

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

- **Def 1(1).** For each statement $x = \mathsf{new}\ c$, we have $\xi(c) \in \Omega(x)$, so $c \dashrightarrow^* x$ by definition.
- **Def 1(2).** For each statement $x = y$, we have $\Omega(y) \subseteq \Omega(x)$, therefore $y \sqsubseteq^* x$ by definition.
- **Def 1(3).** 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.
- **Def 2(1).** It is obvious that $\sqsubseteq^*$ is transitive and reflexive, which is by $\subseteq$ being transitive and reflexive.
- **Def 2(2).** W.l.o.g., for all $x \in \mathtt{VAR}$, $\Omega(x) \neq \emptyset$, since the included type could be from new, or from method parameter passing or return. Given $x \xrightarrow{f} z$, $z' \sqsubseteq^* y$ and $z' \sqsubseteq^* x$, we need to show $y \xrightarrow{f\ *} z$. Let $\xi(c) \in \Omega(z')$, then $\xi(c) \in \Omega(x)$ and $\xi(c) \in \Omega(y)$. Then $\Omega(z) \subseteq \Phi(\xi(c), f)$ by $\xi(c) \in \Omega(x)$ and $x \xrightarrow{f} z$. Therefore we have $y \xrightarrow{f\ *} z$ by the existence of $\xi(c)$.
- **Def 2(3).** For each statement $x = y.f$, given $y \xrightarrow{f\ *} z$, we need to show $z \sqsubseteq^* x$. Equivalently, by definition there exists $o \in \Omega(y)$ such that $\Omega(z) \subseteq \Phi(o_1, f)$. Given the statement $x = y.f$, for all $o' \in \Omega(y)$ $\Phi(o', f) \subseteq \Omega(x)$, therefore by existence of o, we have $\Omega(z) \subseteq \Omega(x)$, the definition of $z \sqsubseteq^* x$.
- **Def 2(4).** The first case that $c \dashrightarrow y$ implies $c \dashrightarrow^* y$ is trivial. Let $c \dashrightarrow^* x$ and $x \sqsubseteq^* y$, we need to show that $c \dashrightarrow^* y$. Since $\xi(c) \in \Omega(x)$ and $\Omega(x) \subseteq \Omega(x)$, we have $\xi(c) \in \Omega(y)$, which is equivalent to $c \dashrightarrow^* y$, as required.

□

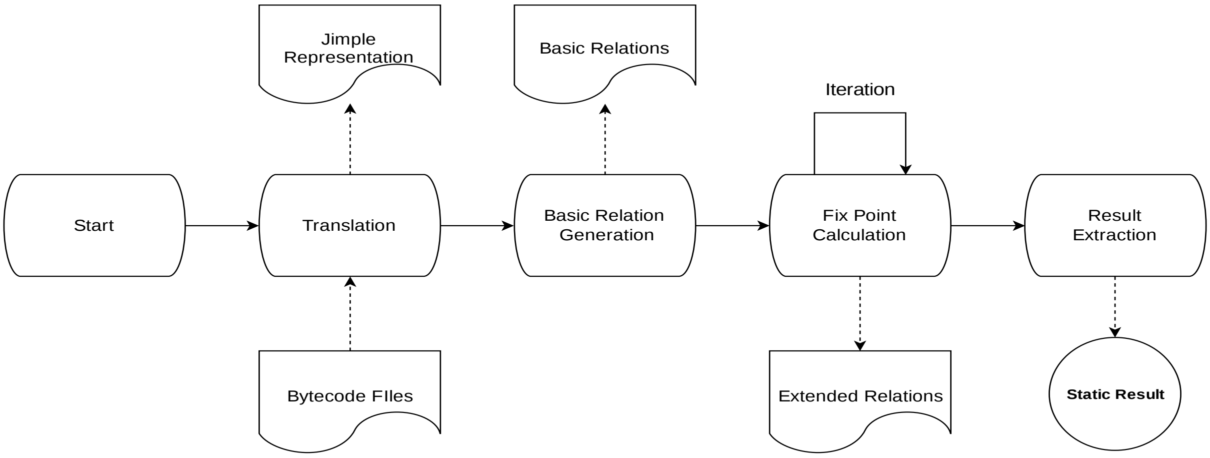


Fig. 8. Process of static analysis tool

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 intermediate representation (IR). We do not take common types (*e.g.* , `int` and `float`) under our consideration, since these are irrelevant to our analysis. Our work consists of two parts. The first part implements the static type flow analysis algorithm introduced in previous sections, based on the information extracted from jimple IR and computes a set of types for each reference variable in the program. The second part instantiates a dynamic profiler that collects type-related information, which is then used to analyze precision and recall of the TFA static analysis algorithm, as well as making comparison to other type analysis algorithms including CHA, VTA and PTA. A more detailed description is given as follows, in Section 3.1 and Section 3.2.

3.1. Static Analysis

Our static analysis tool takes Java bytecode files as input. Any other format of Java code is acceptable provided that it can be translated to jimple representation by Soot (*e.g.* , jar files). The implementation of our static analysis tool can be described in the following four steps.

1. Code translation: Target code is loaded by Soot and translated to jimple IR.
2. Basic relation extraction: We traverse the jimple IR to generate all basic relations by definition.
3. Fixpoint calculation: After basic relations are generated, we use the extended relation rules to perform fixpoint calculation.
4. Result extraction: After all required analysis is done, we extract the set of reaching types as final result.

Note that since our analysis is inter-procedural, type information for a receiver of a call site can be used to resolve potential callee methods, which further help to propagate type information from actual parameters to virtual parameters and from return variables back to the caller statement. Since the additional inter-procedural type flow may incrementally help to enrich type information at call sites, the type flow process is mutually recursive between call sites and defined methods. Fig 8 illustrates the entire process of our static analysis tool. Input data and generated data are represented in dash arrow. The final result is passed on by the “Result Extraction” phase and represented in the final circle.

In order to achieve an acceptable performance, we make conservative treatments on array accesses and library calls. We choose not to analyze method invocation from advanced Java language features such as reflection and JNI, and leave such calls as unresoled. Therefore, our methodology is unsound regarding these features, similar to most existing tools for Java program analysis, to the best of our knowledge. (C.f. an interesting discussion regarding soundiness [?]) We will provide a detailed analysis regarding our conservative approach on the impact to the performance of our implemented TFA algorithm in Section 4.

3.2. Dynamic Profiling

We apply dynamic analysis to collect the grountruth of a program's run-time behaviour, such as the actual type a given variable has during an execution and the actual method being called at a particular program location. To build up a baseline for measuring the precision of TFA, we have implemented a dynamic profiling tool that records the actual types that a variable may have at run-time, in particular when the variable is a receiver at a call site. Based on the Soot framework, we instrument the jimple statements of the target benchmark programs that are subsequently tested in our experiment. After the instrumentation phase, run-time types are collected during the benchmark execution and are dumped to the output. We consider the collected variable-type relation as the groundtruth and compare it with the results generated by our static TFA algorithm, with details presented in section 4.2. In order to achieve effective instrumentation, we consider the following four possible ways on how to instrument a Java program.

Insert First. The type-recording statements are inserted as the first statement of each method block so that the type of `this` reference in that method will be recorded. Note that a receiver is always passed into `this` reference in a callee method, unless the method is static.

Insert Before. New statements are inserted before each call site to collect the receiver type. This is more straightforward than *Insert First*, as we do not need to find the types of `this` variables in callees.

Insert Last. Similar to *Insert First*, except that statements are inserted before the last statement of a method block. We also record the type of `this` reference.

Insert After. ***(our scheme)*** In this way we insert statements right after each call site to record the type of the receiver, which is similar to *Insert Before*. We choose to adopt this scheme and the reason is given as follows.

According to the Java specification, the first statement of each constructor should be either another constructor of its own or its super class. Therefore, we would get a *JVM voilation* error if we instrument a statement before the first statement in a constructor. We illustrate this problem in an example given in Listing 2. Due to this restriction, both *Insert First* and *Insert Before* schemes cannot be applied. For the remaining to schemes, we choose *Insert After* over *Insert Last* for the reason that a last statement could be appeared in many different kinds like a return statement, an exception throwing statement, or even an application terminating invocation. It will take more time to tackle this problem if we use the *Insert Last* scheme to implement our profiler. More details are displayed in an example in Listing 3. We can easily insert at line 4 and line 8, for an exception throwing statement and a return statement, respectively. However, it requires significant time to recognize line 7 as a last statement of method $m1$ since it will terminate the whole application. As a result, we use *Insert After* as our implementation manner so that a type can be more easier and straightforwardly collected and bundled with its receiver. As an example, the code before and after instrumentation are shown in Listing 4 and Listing 5, respectively. For Listing 5, 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. In this example, the collected unique representation is *A:m1:b:B:m2:4*, which stands for b calls method $m2$ of class B, at line 4 within method $m1$ of class A. It also shows that b is of type B at this program location.

```
class A {
  public A() {
    //insert statements here will violate JVM specification
    super();     //invoke super class constructor
  }
  public A(int i) {
    //insert statements here will violate JVM specification
    this();     //inovke another constructor of its own
  }
}
```

Listing 2: Java specification on constructor

```
class A {
  public Obejct m1(int i) {
    if(i>0) {
      // insert here, right before an exception throwing statement
      throw new Exception(‘‘i>0’’);
    }
    exit(); // this invocation terminates application
    // insert here, right before a return statement
    return null;  // insert before this statement
  }

  public void exit() {
    System.exit(0);  // this invocation terminates application
  }
}
```

Listing 3: Example code of *Insert Last*

```
class A {
  public void m1() {
    B b = new B();
    b.m2();     //invocation here
  }
}
```

Listing 4: Example code before our instrumentation

```
class A {
  public void m1() {
    B b = new B();
    b.m2();     //invocation here
    String record = RecordUtils.id(invokeExpression, b);
    RecordUtils.record(record);
  }
}
```

Listing 5: Example code after our instrumentation

4. Evaluation

We evaluate our approach by measuring its performance on SPECjvm2008 [?], which consists of 12 benchmark programs in total. We conduct our experiment 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 Class Hierarchy Analysis (CHA) , Variable Type Analysis (VTA) and context-insensitive points-to analysis (PTA). Among these compared approaches, we use the hierarchy structure provided in Soot framework to implement CHA. The VTA and the points-to analysis are available from the Soot framework, which are both implemented by Soot team. We discover that since the VTA implementation in Soot is only an optional subprocess used to generat callgraph for points-to analysis, only the generated call graph is accessible by developer, and the general application interface of VTA is invisible to normal Soot users. Due to this restriction, we are unable to apply VTA as an independent process to measure its performance on the SPECjvm2008 benchmark programs. Nevertheless, we manage to compare precision and recall of our results with the call graph that is generated by the VTA subprocess, as shown in Figure 10 and Figure 11.

Benchmark	$\mathbf{T}_{\textsf{CHA}}(s)$	$\mathbf{T}_{\textsf{PTA}}(s)$	$\mathbf{T}_{\textsf{TFA}}(s)$	$\mathbf{R}_{\dashrightarrow}$	$\mathbf{R}_{\sqsubseteq}$	$\mathbf{R}_{\rightarrow}$	$\mathbf{R}_{total}$
check	0.35	69.32	3.52	8718	15599	10089	34406
compiler	0.27	70.34	1.14	7781	11277	1990	21048
compress	0.28	71.73	1.09	7428	10947	1876	20251
crypto	0.25	69.22	1.13	7575	11220	1846	20641
derby	0.31	70.79	1.38	7979	11450	2034	21463
helloworld	0.25	70.44	1.25	7230	10543	1828	19601
mpegaudio	0.25	69.13	1.04	7279	10612	1830	19721
scimark	0.24	71.01	1.15	7451	11344	1845	20640
serial	0.43	69.20	1.07	7683	10977	1886	20546
startup	0.47	134.09	1.46	11597	17870	2651	32118
sunflow	0.25	71.26	1.07	7245	10560	1830	19635
xml	0.45	140.72	1.24	7837	11673	2016	21526

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

Fig. 9. Runtime cost with different analysis

The choice of comparing with the context-insensitive points-to analysis implementation in Soot is due to our approach also being context-insensitive, thus the results from both approaches are comparable. In particular, the iterative algorithm for points-to analysis provided by the Soot team adopts a more space-efficient *bit vector* data structure to avoid *Out-Of-Memory* error, as the points-to computation usually consumes more memory than TFA. For the implementation of TFA, we apply the usual data structure (*i.e.* *HashSet*).

In the following tables listed in Figure 9, Figure 10 and Figure 11, we use the keywords **CHA**, **VTA**, **PTA** and **TFA** to refer to the results from class hierarchy analysis, variable type analysis, points-to analysis and type flow analysis, respectively. During the evaluation the following two research questions are addressed.

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

We try to provide answers to these two questions as discussed in section 4.1 and section 4.2, respectively.

4.1. Analysis on Performance

We have executed all SPECjvm2008 benchmark programs 10 times with the CHA, PTA and TFA algorithms. Then we calculate 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 Figure 9. We also list the sizes of the generated relations (*i.e.* , the type flow relation '$\dashrightarrow$', variable partial order '$\sqsubseteq$' and the field access relation '$\longrightarrow$'), which provide an estimation of input size for the problems we are treating. The result confirms that our approach consumes more time than CHA, which is reasonable, as CHA does not apply any propagation on types, but only analyzes the class and interface hierarchy structure.[4] TFA is in general more efficient than points-to analysis. In general, the runtime cost of TFA increases on benchmarks with larger size of generated relations as well as higher relational complexity, as most of the time in TFA is spent on calculating a fixpoint (*e.g.* , on benchmark *check* the number of generated relations reach to $34,406$, on which TFA spends the most amount of time($3.52s$) among all benchmarks). We conjecture that PTA is slower than TFA because it requires extra time for maintaining and updating a heap abstraction.

[4] Note that we have implemented the current CHA algorithm by using the class hierarchy structure provided in the Soot framework. The default CHA implementation in Soot runs much slower, as shown in our earlier experiment result in [**?**].

Benchmark	**Dynamic**	**CHA**			**VTA**			**PTA**			**TFA**		
		TP	*FN*	*Recall*	*TP*	*FN*	*Recall*	*TP*	*FN*	*Recall*	*TP*	*FN*	*Recall*
check	149	149	0	1.000	116	33	0.779	116	33	0.779	145	4	0.973
compiler	487	486	1	0.998	297	190	0.610	300	187	0.616	466	21	0.957
compress	351	351	0	1.000	247	104	0.704	313	38	0.892	313	38	0.892
crypto	449	449	0	1.000	279	170	0.621	349	100	0.777	406	43	0.904
derby	583	583	0	1.000	352	231	0.604	433	150	0.743	549	34	0.942
helloworld	21	21	0	1.000	20	1	0.952	20	1	0.952	21	0	1.000
mpegaudio	254	254	0	1.000	213	41	0.839	232	22	0.913	246	8	0.969
scimark	377	377	0	1.000	283	94	0.751	349	28	0.926	367	10	0.973
serial	451	451	0	1.000	298	153	0.661	340	111	0.754	399	52	0.885
startup	1344	1337	7	0.995	1048	296	0.780	1117	227	0.831	1096	248	0.815
sunflow	225	225	0	1.000	200	25	0.889	200	25	0.889	214	11	0.951
xml	513	513	0	1.000	315	198	0.614	407	106	0.793	459	54	0.895

[1] **Dynamic** denotes the number of dynamic records.

[2] *TP*, *FN* denote true positive, false nagative of different algorithms, respectively.

Fig. 10. Recall on different analysis

4.2. Analysis on Accuracy

In this section we address the second question about accuracy in two perspectives: recall and precision. Firstly, the definitions are given in Equation 1 and Equation 2, respectively.

$$recall = \frac{TP}{TP + FN} \tag{1}$$

$$precision = \frac{TP}{TP + FP} \tag{2}$$

Here, TP stands for *true positive* which refers to the number of call edges identified by the analysis which also happens during program execution. FN stands for *false negative*, which is the number of call edges missed by the analysis. FP (*false positive*) refers to the number of call edges erroneously generated by the analysis. In general, a sound analysis should catch all existing runtime type information, *i.e.* , the *recall* should be 100%. On the other hand, *precision* is defined as percentage of reported types that are correctly generated by the analysis. In general, given *recall* fixed, a more precise algorithm should generate a smaller set of types for each calling variable. We present recall and precision of the given approaches in Figure 10 and Figure 11, respectively.

Regarding soundness, we use the call edges generated by the dynamic profiler (described in Section 3.2) as baseline, shown at the **Dynamic** columns of Figure 10 and Figure 11. In general, CHA is able to achieve nearly 100% recall, but VTA, TFA and PTA all fail to achieve 100% recall except for the `helloworld` program. TFA is able to achieve above 90% recall on average, higher than VTA and PTA (20% more recall on about half of the benchmark programs). Regarding precision, TFA archieves about 80% on average, lower than VTA and PTA, but in general higher than CHA.

Given that TFA achieves a higher recall than VTA and PTA, we have identified a few possible reasons. Firstly, VTA and PTA implemented by the Soot team have applied an on-the-fly reachability analysis which rules out a portion of methods which are reported as unreachable. Therefore the unreachable methods would not be analyzed, and variables receiving their type from JDK would be assigned with possible types based on the preprocessing from Soot. For TFA, we have implemented a conservative approach on JDK library functions which always returns the predefined types of the API together with all their subtypes. We also implemented a conservative array analysis which may contribute to higher recall as well as lower precision. We also note that we have not implemented any special treatment regarding advanced Java language features including Java Reflection API and Java Native Interface calls, as well as more detailed analysis in the JDK library and side effects produced by callbacks from invisible codes (*e.g.* from JDK library and reflective calls), which all contribute to the existence of FN instances, so that TFA does not produce a sound result

Benchmark	**Dynamic**	**CHA**			**VTA**			**PTA**			**TFA**		
		TP	*FP*	*Pre*	*TP*	*FP*	*Pre*	*TP*	*FP*	*Pre*	*TP*	*FP*	*Pre*
check	149	149	22	0.871	116	6	0.951	116	6	0.951	145	46	0.759
compiler	487	486	427	0.532	297	0	1.000	300	0	1.000	466	123	0.791
compress	351	351	86	0.803	247	0	1.000	313	7	0.978	313	50	0.862
crypto	449	449	169	0.727	279	4	0.986	349	5	0.986	406	123	0.767
derby	583	583	185	0.759	352	0	1.000	433	19	0.958	549	112	0.831
helloworld	21	21	2	0.913	20	0	1.000	20	0	1.000	21	1	0.955
mpegaudio	254	254	59	0.812	213	0	1.000	232	6	0.975	246	33	0.882
scimark	377	377	135	0.736	283	0	1.000	349	54	0.866	367	86	0.810
serial	451	451	6684	0.063	298	0	1.000	340	2336	0.127	399	70	0.851
startup	1344	1337	1700	0.440	1048	837	0.556	1117	1021	0.522	1096	989	0.526
sunflow	225	225	50	0.818	200	0	1.000	200	0	1.000	214	23	0.903
xml	513	513	322	0.614	315	4	0.987	407	60	0.872	459	122	0.790

[1] **Dynamic** denotes the number of dynamic records.

[2] *TP*, *FP* and *Pre* denote true positive, false nagative and presicion of different algorithms, respectively.

Fig. 11. Precision of different analysis

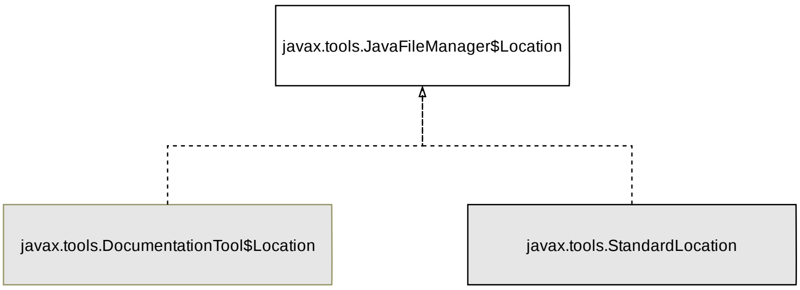

javax.tools.JavaFileManager$Location
javax.tools.DocumentationTool$Location
javax.tools.StandardLocation

Fig. 12. A simple hierarchy in JDK

(*i.e.* 100% recall) on most benchmark programs.[5] For a more detailed explanation regarding the conservative treatment implemented in TFA, we refer to Section 4.3.

We have also investigated why CHA also missed 8 call edges on the benchmarks `compiler` and `startup`. The one missed on `compiler` is caused by the incomplete default hierarchy structure provided by the Soot framework. As an example, we put on a hierarchy structure of $javax.tools.JavaFileManager\$Location$ in JDK, as shown in Figure 12. Note that variable x is declared of type $javax.tools.JavaFileManager\$Location$ in benchmark `compiler`. However, the hierarchy structure provided by the Soot framework does not properly include $javax.tools.DocumentationTool\$Location$ or $javax.tools.StandardLocation$ as subclassses of $javax.tools.JavaFileManager\$Location$. Hence, these 2 types are missed by CHA for variable x. The other 7 missed types in CHA are due to that the Soot framework mistakenly changes the name of a few temporary variables (*e.g.* , variable $68 is changed to $58), which can be detected by inspecting the jimple representation of the benchmark `startup`.

For a comparison of the algorithms regarding precision, the experiment result has shown that in general TFA achieves higher precision than CHA (except for the benchmark `check` on which TFA's conservative treatment on JDK library has generated 40 more FPs), since CHA simply includes all subclasses of a declared class to the result. At the same time, TFA produces lower precision than VTA and PTA due to its decisive trade-off between recall and precision. The imprecision is mainly due to the over approximation on library invocation and array (we defer a detailed discussion to section 4.3). These conservative treatment has raised up recall but lowered precision at the same time. For example, on benchmark `compiler`, the recall of TFA achieves 0.957, which is much higher than VTA(0.610) and PTA(0.616), as TFA catches more call edges

[5] As we mentioned earlier, most existing tools for Java program analysis are not sound, but soundy [**?**].

Benchmark	**Dynamic**	**CHA**			**VTA**			**PTA**			**TFA**		
		TP	*FN*	*Recall*	*TP*	*FN*	*Recall*	*TP*	*FN*	*Recall*	*TP*	*FN*	*Recall*
check	149	149	0	1.000	116	33	0.779	116	33	0.779	89	60	0.597
compiler	487	486	1	0.998	297	190	0.610	300	187	0.616	361	126	0.741
compress	351	351	0	1.000	247	104	0.704	313	38	0.892	276	75	0.786
crypto	449	449	0	1.000	279	170	0.621	349	100	0.777	294	155	0.655
derby	583	583	0	1.000	352	231	0.604	433	150	0.743	411	172	0.705
helloworld	21	21	0	1.000	20	1	0.952	20	1	0.952	16	5	0.762
mpegaudio	254	254	0	1.000	213	41	0.839	232	22	0.913	213	41	0.839
scimark	377	377	0	1.000	283	94	0.751	349	28	0.926	332	45	0.881
serial	451	451	0	1.000	298	153	0.661	340	111	0.754	335	116	0.743
startup	1344	1337	7	0.995	1048	296	0.780	1117	227	0.831	804	540	0.598
sunflow	225	225	0	1.000	200	25	0.889	200	25	0.889	192	33	0.853
xml	513	513	0	1.000	315	198	0.614	407	106	0.793	377	136	0.735

Fig. 13. Recall without library approximation

Benchmark	**Dynamic**	**CHA**			**VTA**			**PTA**			**TFA**		
		TP	*FP*	*Pre*	*TP*	*FP*	*Pre*	*TP*	*FP*	*Pre*	*TP*	*FP*	*Pre*
check	149	149	22	0.871	116	6	0.951	116	6	0.951	89	6	0.937
compiler	487	486	427	0.532	297	0	1.000	300	0	1.000	361	38	0.905
compress	351	351	86	0.803	247	0	1.000	313	7	0.978	276	31	0.899
crypto	449	449	169	0.727	279	4	0.986	349	5	0.986	294	17	0.945
derby	583	583	185	0.759	352	0	1.000	433	19	0.958	411	37	0.917
helloworld	21	21	2	0.913	20	0	1.000	20	0	1.000	16	0	1.000
mpegaudio	254	254	59	0.812	213	0	1.000	232	6	0.975	213	18	0.922
scimark	377	377	135	0.736	283	0	1.000	349	54	0.866	332	66	0.834
serial	451	451	6684	0.063	298	0	1.000	340	2336	0.127	335	30	0.918
startup	1344	1337	1700	0.440	1048	837	0.556	1117	1021	0.522	804	792	0.504
sunflow	225	225	50	0.818	200	0	1.000	200	0	1.000	192	12	0.941
xml	513	513	322	0.614	315	4	0.987	407	60	0.872	377	84	0.818

Fig. 14. Precision without library approximation

than VTA(169) and PTA(166). As a result, TFA generates more FP than VTA(123) and PTA(123), with a lower precision of 0.791. Focusing on the benchmark `serial` and `startup` as shown in Figure 11, we notice that PTA has generated lots of FPs, which lowers its precision to 0.127 and 0.522. This is caused by Soot's conservative treatment on Java *Exception* and its preprocessing on some specific library calls.

Due to the conservative approach on JDK library function, the TFA analysis achieves higher recall with the price of lower precision in comparison to VTA and PTA. Note that in theory context insensitive TFA is equivalent to context insensitive PTA and is more precise than VTA, as sudied in Section 2. In order to make the implementations more comparable, we modify the implementation of TFA by removing the library call approximation to JDK. As shown in Figure 13 and Figure14, without the conservative treatment on JDK library, in general TFA has a comparable precision and recall to VTA and PTA on the given benchmark suite. To explain the reason that VTA and PTA marginally outperform TFA in a few benchmarks, we refer to the optimization processes implemented by the Soot team on array operations and certain reflective calls that are exploited by the VTA and PTA processes.

4.3. A Detailed Analysis on TFA's FP and FN

In this section we briefly review the factors that hamper the performance of our type flow analysis.

Reflective calls. Reflection is an advanced feature in the Java programming language which provides the ability to inspect and manipulate a Java class at runtime. It brings in extra complexity to programs and its behaviour is hard to predict statically. In Listing 6 we show a sample code using reflection in one of the benchmark programs to discuss how reflection works and what our treatment is. Note that this is an example we have simplified from the real code in order to concentrate on the main functionality of reflection. We discuss in three cases.

- Object Creation. From line 6 to line 10, a new obejct of type *SPECJVMBenchmarkBase* is created by method *newInstance*() on variable *c* at line 10, where *c* is an object of *Constructor* type and it refers to a specific constructor of class *SPECJVMBenchmarkBase*. In the TFA analysis, we discard the type information of the new object in such a case because it is usually difficult to statically identify which constructor is invoked. For example, At line 6, if statement *Class.forName*() receives a string argument from the outside such as file input and user input, it is usually not easy to find out the real type of *benchmarkClass*. As a result, the run time type of *c* and *benchmark* cannot be identified.
- 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 this method by the method *invoke*() (line 17). This effect is similar to a normal invocation at line 12. The invocation receiver is passed to the first argument of *invoke*(). In a case of static invocation, a **null** reference is passed to the first argument. We do not consider this way of reflective method invocation in TFA, for the same reason as we discuss about object creation.
- Field Modification. The way to change a field using reflection is similar to processing method invocation. The line $24-25$ illustrate changing value of field f on object *benchmark* to a new object. We do not handle this effect in TFA, either.

```java
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 6: Example code of reflection

JNI calls. 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 7. 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 implemented in a different program language. 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 the invocation $a.m()$ at line 8 is not analyzed by TFA.

```
 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 7: Example code of JNI

Library call. Library calls included in an application (*e.g.* , JDK libraries, third-party libraries) are usually in pre-compiled bit code format, and their source code is often large and not made explicitly available. Listing 8 shows a common case of JDK invocation. We do not analyze the detailed 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 8, $sb2$ will receive $\{StringBuilder,\ any\_subtype\_of\_StringBuilder\}$ as the set of reaching types under this 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 8: Example of JDK library call

Array approximation. We perform a conservative treatment on array accesses in the way that all type information that flows to one member of an array flows to all members of that array. Codes in Listing 9 are used to explain how this approximation works. In this example, type information of A and B flow to the first member and the second member of array arr, respectively. We treat in the way that both A and B flow to the array arr, so that loadding an element of arr receives both types ($i.e.$ b receives $\{A, B\}$ as the set of reaching types, which is the same set of types that array arr can access).

```
public void m() {
     Object[] arr = new Object[2]{};
     arr[1] = new A();
     arr[2] = new B();
     Object b = arr[1];  <——
}
```

Listing 9: Example of array access

Callbacks. A callback function is known as a "call-after" function, which is widely used in object-oriented styled programming. By passing an argument to a method, the receiver may be used inside the callee to trigger a callback invocation. In functional language a callback funtion can be passed as argument directly. Listing 10 illustrates the callback mechanism, where function *callback*() of class *B* is invoked after the invocation of *trigger*() of class *A*. Since we do not analyze the effect of reflection calls, JNI calls and library calls, 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.

```
class A {
    public void trigger(B b) {
        b.callback();  // here the callback function actually executes
    }
}

class B {
    public void callback() {...}
    public void static main() {
        A a = new A();
        B b = new B();
        a.trigger(b);
    }
}
```

Listing 10: Callback mechanism

Exception handling. PTA implemented by Soot conservatively add all the subclass of *Throwable* to the possible types of variable t if method $m()$ is reachable. Relevant code is shown in Listing 11. Since *Throwable* is the superclass of all errors and exceptions in Java. These result set could be pretty large. In general it would reach to more than 200. That means if we only have a single dynamic record for this invocation statement, the precision will be reduced significantly.

```
class A {
    void m() {
        try {
            ...
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

Listing 11: Java Exception

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.

Instrumentation for Java bytecode can be performed either statically or dynamically. Traditional instrumentation is executed statically at most time, which suffers from the limitation that dynamically generated code or downloaded code will be skiped from instrumentation. Walter et al. [?] implemented an framework called FERRARI which can perform instrumentation in both ways. However, our dynamic profiler is implemented using statically instrumentation. Hence, dynamically generated code can not be instrumented. We do not implemented dynamic instrumentation yet, but note that all the benchmarks code we tested are statically asscessible (*i.e.* , none of codes are dynamically generated or downloaded).

6. Conclusion and future work

In this paper we have proposed Type Flow Analysis (TFA), a new semantics that statically determines reaching types of variables and constructs a relatively precise call graph edges for Object-Oriented programming languages in an efficient way. We have proved that our method is as precise as subset-based points-to analysis, regarding type related information. The computation of TFA is purely relational, since it do not require a heap abstraction that is needed by a 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 available in Soot regarding precision and recall. The result have shown that our method achieved promising result and it is confirmed that in general TFA is applicable in the type resultion problem for real world applications.

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

Acknowlegements The authors thank Bernhard Scholz for his guidance in our experiment regarding JVM configurations. We also thank anonymous reviewers of ICFEM 2019 for their help suggestions.