A Relational Static Semantics for Call Graph Construction

Xilong Zhuo and Chenyi Zhang

Jinan University

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, in practice, our method consumes lower time and space usage, as supported by the experimental results.

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) [6, 7], Rapid Type Analysis (RTA) [4] and Variable Type Analysis (VTA) [18], 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 [3] 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 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

```
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
$A\ x = \text{new}\ A()$	$x \leftarrow A$
$B\ b = \text{new}\ B()$	$b \leftarrow B$
$A\ y = \text{new}\ A()$	$y \leftarrow A$
$C\ c = \text{new}\ C()$	$c \leftarrow C$
$x.f = b$	$A.f \leftarrow b$
$y.f = c$	$A.f \leftarrow c$
$z = x.m()$	$A.m.this \leftarrow x$ $A.m.return \leftarrow A.f$ $z \leftarrow A.m.return$

Fig. 2. VTA facts on the example

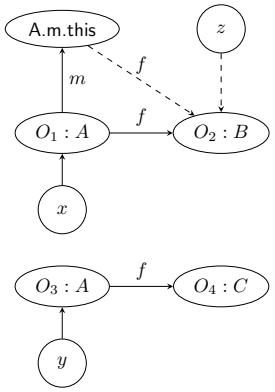

Fig. 3. Points-to results on the example

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, 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 [3] creates a heap abstraction of four objects (shown on line $1 - 4$ of Figure 1 as well as

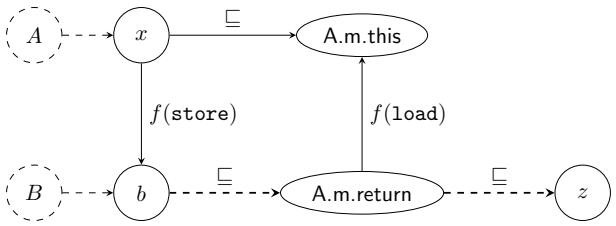

Fig. 4. Type Flow Analysis for variable z in the example

in 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, besides a binary value flow relation $\sqsubseteq$ on the variable domain $\texttt{VAR}$, where $x \sqsubseteq y$ denotes all types that flow to x also flow to y, we also build a ternary field store relation $\rightarrow \subseteq \texttt{VAR} \times \mathcal{F} \times \texttt{VAR}$ 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 [15, 10], object-sensitivity [12, 16, 19] and type-sensitivity [16]), a context-sensitive type flow analysis will only require to consider contexts on variables, which is left for future work. The context-insensitive type flow analysis has been implemented in the Soot framework [1], and the implementation has been tested on a collection of benchmark programs from SPECjvm2008 [2] and DaCapo [5]. The initial experimental result has shown that TFA consumes similar or less runtime than CHA [6], but has precision comparable to that of a points-to analysis.

2 Type Flow Analysis

We define a core calculus consisting of most of the key object-oriented language features, as shown in Figure 5. 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. $fields$ maps class names to its fields, $methods$ maps class names to its 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 an object of any of its subclass at runtime. Write $\mathcal{C}$ for the set of classes, `OBJ` for the set of objects, $\mathcal{F}$ for the set of fields and `VAR` for the set of variables that appear in a program.

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

Fig. 5. Abstract syntax for the core language.

In this simple language we do not model common types (e.g., int and float) that are irrelevant to our analysis, and we focus on the reference types which form a class hierarchical structure. Similar to Variable Type Analysis (VTA), 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 the 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 methods and classes.

We enrich the variable type analysis with the new type flow analysis by using three relations, 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) We have the following base facts for the three relations.*

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 from variable x and field f one may access variable y.[1] These three relations are then extended by the following rules.

Definition 2. *(Extended Relations)*

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

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

The final relations are the least relations that satisfy constraints of Definition 2. Comparing to VTA [18], we do not have field reference $c.f$ for each class c defined in a program. Instead, we define a relation that connects the two variable names and one field name. 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 classic subset-based points-to rules for our language in Figure 6, where $param(type(o), m))$, $this(type(o), m))$ and $return(type(o), m)$ refer to the formal parameter, this reference and return variable of the method m of the class for which object o is declared, respectively.

To this end we present the first result of the paper, which basically 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 analysis, for all variables* x *and classes* c*,* $c \dashrightarrow^* x$ *iff there exists an object abstraction* o *of* c *such that* $o \in \Omega(x)$*.*

Proof. (sketch) For a proof sketch, first we assume 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 classes 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

[1] 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 $c.f$.

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

Fig. 6. Constraints for points-to analysis.

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. □

Lemma 1. *In a context-insensitive intraprocedural analysis where each class c only syntactically appears once in the form of* $\mathsf{new}\ c$*, 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 the points-to constraints define the smallest model (Ω, Φ) with $\Omega : \mathtt{VAR} \to \mathtt{OBJ}$ and $\Phi : \mathtt{OBJ} \times \mathcal{F} \to \mathcal{P}(\mathtt{OBJ})$, 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 of points-to 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 $Reaches(x) = \{c \mid c \dashrightarrow^* x\}$, and assume a bijection $\xi : \mathcal{C} \to \mathtt{OBJ}$ that maps each class c to the unique object o that is defined (and $type(o) = c$). Then we construct a function $Access : \mathcal{C} \times \mathcal{F} \to \mathcal{P}(\mathcal{C})$ and show that $(\xi(Reaches), \xi(Access))$ satisfies the points-to constraints. Define $Access(c, f) = \{c' \mid x \xrightarrow{f}{}^{*} y \wedge c \in Reaches(x) \wedge c' \in Reaches(y)\}$. We prove the following cases according to the top four points-to constraints in Figure 6.

- For each statement $x = \mathsf{new}\ c$, we have $\xi(c) \in \xi(Reaches(x))$;
- For each statement $x = y$, we have $Reaches(y) \subseteq Reaches(x)$ and $\xi(Reaches(y)) \subseteq \xi(Reaches(x))$;
- For each statement $x.f = y$, we have $x \xrightarrow{f} y$, then by definition for all $c \in Reaches(x)$, and $c' \in Reaches(y)$, we have $c' \in Access(c, f)$, therefore $\xi(c') \in \xi(Reaches(y))$ we have $\xi(c') \in \xi(Access(\xi(c), f))$.
- For each statement $x = y.f$, let $c \in Reaches(y)$, we need to show $\xi(Access(c, f)) \subseteq \xi(Reaches(x))$, or equivalently, $Access(c, f) \subseteq Access(x)$. Let $c' \in Access(c, f)$, then by definition, there exist z, z' such that $c \in Reaches(z)$, $c' \in Reaches(z')$

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 6, 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 $\to$ that are just defined from the above.

- For each statement $x = \mathsf{new}\ c$, 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. □

3 Implementation and Optimization

The analysis algorithm is written in Java, and is implemented in the Soot framework [1], the most popular static analysis framework for Java. The three base relations (i.e., $\dashrightarrow$, $\sqsubseteq$ and $\to$) of Definition 1 are extracted from Soot's intermediate representation and the extended relations (i.e., $\dashrightarrow^{*}$, $\sqsubseteq^{*}$ and $\to^{*}$) of Definition 2 are then computed considering the mutual dependency relations between them. Since we are only interested in reference types, we do not carry out analysis on basic types such as boolean, int and double. We also do not consider more advanced Java features such as functional interfaces and lambda expressions, as well as usages of Java Native Interface (JNI), nor method calls via Java reflective API. We have not tried to apply the approach to Java libraries, all invocation of methods from JDK are treated as end points, thus all possible call back edges will be missed in the analysis. Array accesses are treated conservatively—all type information that flows to one member of a reference array flows to all members of that array, so that only one node is generated for each array.

Since call graph information may be saved and be used for subsequent analyses, we propose the following two ways to reduce storage for computed result. If a number of variables are similar regarding type information in a graph representation, they can be merged and then referred to by the merged node.

1. If $x \sqsubseteq^* y$ and $y \sqsubseteq^* x$, we say x and y form an *alias pair*, written $x \sim y$. Intuitively, $\texttt{VAR}/\sim$ is a partition of VAR such that each $c \in \texttt{VAR}/\sim$ is a strongly connected component (SCC) in the variable graph edged by relation $\sqsubseteq$, which can be quickly collected by using Tarjan's algorithm [20].
2. A more aggressive compression can be achieved in a way similar to bisimulation minimization of finite state systems [9, 13]. Define $\approx \subset \texttt{VAR} \times \texttt{VAR}$ such that $x \approx y$ is symmetric and if
   - for all class c, $c \dashrightarrow^* x$ iff $c \dashrightarrow^* y$, and
   - for all $x \xrightarrow{f\ *} x'$ there exists $y \xrightarrow{f\ *} y'$ and $x' \approx y'$.

It is straightforward to see that $\approx$ is a more aggressive merging scheme.

Lemma 2. *For all $x, y \in \texttt{VAR}$, $x \sim y$ implies $x \approx y$.*

We have implemented the second storage minimization scheme by using Kanellakis and Smolka's algorithm [9] which computes the largest bisimulation relation for a given finite state labelled transition system. In our interpretation, the variables are treated as states and the field access relation is treated as the state transition relation. The algorithm then merges equivalent variables into a single group. As a storage optimization process, this implementation has been tested and evaluated in the next section.

4 Experiment and Evaluation

We evaluate our approach by measuring its performance on 13 benchmark programs. Among the benchmark programs, *compress*, *crypto* are from the SPECjvm2008 suite [2], and the other 11 programs are from the DaCapo suite [5]. We randomly selected these test cases from the two benchmark suites, in order to test code bases that are representative from a variety of sizes. All of our experiments were conducted on a Huawei Laptop equipped with Intel i5-8250U processor 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 [11] that are implemented by the Soot team. We use Soot as our basic framework to extract the SSA based representation of the benchmark code. We also generate automata representation for the resulting relations which can be visualized in a subsequent user-friendly manual inspection. The choice of the context-insensitive points-to analysis is due to our approach also being context-insensitive, thus the results will be comparable. 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 three research questions are addressed.

bench	$\mathbf{T_{CHA}}(s)$	$\mathbf{T_{PTA}}(s)$	$\mathbf{T_{TFA}}(s)$	$\mathbf{R}_{\dashrightarrow}$	$\mathbf{R}_{\sqsubseteq}$	$\mathbf{R}_{\rightarrow}$
compress	0.02	0.12	0.02	87	202	24
crypto	0.01	0.12	0.03	94	226	18
bootstrap	23.74	34.13	0.03	191	453	17
commons-codec	0.009	0.12	0.13	306	3324	49
junit	24.45	34.24	0.18	1075	5772	241
commons-httpclient	0.009	0.12	0.36	2423	8511	521
serializer	22.60	32.68	1.71	3006	17726	331
xerces	21.69	34.83	3.74	12590	72503	2779
eclipse	21.95	42.64	1.68	7933	37435	1620
derby	22.77	48.82	18.09	20698	191854	5386
xalan	78.40	-	42.11	32971	162249	3696
antlr	44.20	-	3.96	16117	76741	3879
batik	45.84	-	6.38	29409	122534	6039

Fig. 7. Runtime cost with different analysis

RQ 1: How efficient is our approach compared with the traditional class hierarchy analysis and points-to analysis?

RQ 2: How accurate is the result of our approach when comparing with the other analyses?

RQ 3: Does our optimization (or minimization) algorithm achieve significantly reduce storage consumption?

4.1 RQ1: Efficiency

To answer the first research question, we executed each benchmark program 10 times with the CHA, PTA and TFA algorithms. We calculated the average time consumption as displayed at the left three columns of the Table in Figure 7. The sizes of each relation that our approach generated (i.e., the type flow relation '$\dashrightarrow$', variable partial order '$\sqsubseteq$' and the field access '$\rightarrow$') are counted, which provides an estimation of size for the problem we are treating. One may observe that when the problem size increases, the execution time of the our algorithm also increases in a way similar to CHA, though in general the runtime of CHA is supposed to grow linearly in the size of a program. The reason that TFA sometimes outperforms CHA may be partially due to the size of the intrinsic complexity of the class and interface hierarchical structure that a program adopts. TFA is in general more efficient than the 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 requires extra time for maintaining and updating a heap abstraction.

Taking a closer look at the benchmark *bootstrap*, CHA and PTA analyze the benchmark using about 23.74 and 34.13 seconds, respectively. As TFA only generated 661 relations, the analysis only takes 0.03 second. There are 3 benchmarks, *xalan, antlr, batik*, that cannot be analyzed by PTA, marked as "-" in

bench	$\mathbf{CS_{base}}$	$\mathbf{CS_{CHA}}$	$\mathbf{CS_{PTA}}$	$\mathbf{CS_{TFA}}$
compress	153	160	18	73
crypto	302	307	62	121
bootstrap	657	801	891	328
commons-codec	1162	1372	270	554
junit	3196	17532	11176	1197
commons-httpclient	6817	17118	567	2928
serializer	4782	9533	1248	1756
xerces	24579	56252	10631	8120
eclipse	23607	95073	70016	9379
derby	69537	180428	85212	16381
xalan	57430	155866	-	18669
antlr	62007	147014	-	17177
batik	56877	235071	-	20901

Fig. 8. Call sites generated by different analyses

the table in Figure 7. In these cases the points-to analysis has caused an exception called "OutOfMemoryError with JVM GC overhead limit exceeded", as the JVM garbage collector is taking an excessive amount of time (by default 98% of all CPU time of the process) and recovers very little memory in each run (by default 2% of the heap).

4.2 RQ2: Accurancy

We answer the second question by considering the number of generated call sites as an indication of accuracy. In type flow analysis, a method call $a.m()$ is resolved to $c.m()$ if class c is included in a's reaching type set and method $m()$ is defined for c. In general, a more accurate analysis often generates a smaller set of types for each calling variable, resulting fewer call edges in total in the call graph. The table included in Figure 8 displays the number of call sites generated by different analyses. We also include the base call site counting, i.e., the number of call sites syntactically written in the source code, as the baseline at the $\mathbf{CS_{base}}$ column. In comparison to CHA, our approach has reduced a significant amount of call site edges. Comparing to other two analyses, the number of call edges resolved by TFA are often larger than PTA and smaller than CHA on the same benchmark. The difference may be caused by our over approximation on analyzing array references, as well as the existence of unsolved call edges from e.g. JNI calls or reflective calls.

4.3 RQ3: Optimization

We apply bisimulation minimization to merge nodes that have the same types as well as accessible types through fields. Thus we can reduce the space consumption when storing the result for subsequent analysis processes. Regarding the third research question, we calculated the number of "effective" nodes before and

bench	$\text{Node}_{\text{origin}}$	Node_{opt}	Reduce	Time(s)
compress	205	130	36.59%	0.008
crypto	312	158	49.36%	0.010
bootstrap	514	279	45.72%	0.019
commons-codec	1742	886	49.14%	0.202
junit	5859	3243	44.65%	2.269
commons-httpclient	9708	5164	46.81%	4.651
serializer	9600	6668	30.54%	3.198
xerces	41634	-	-	-
eclipse	34631	-	-	-
derby	112265	-	-	-
xalan	103697	-	-	-
antlr	56589	-	-	-
batik	89336	-	-	-

Fig. 9. Optimization result

after optimization process. Besides, time consumption is another factor that we consider. The results are shown in the table in Figure 9. We evaluated our optimization algorithm on all benchmarks 10 times and successfully executed 7 out of the 13 benchmarks. For those succeeded, the algorithm has reduced space usage by about 45% on average, with a reasonable time consumption. The other 6 benchmarks cannot be due to the exception "OutOfMemoryError with JVM GC overhead limit exceeded" being thrown, for which we leave the symbol "-" accordingly in the table.

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) [6, 7], Rapid Type Analysis (RTA) [4] and Variable Type Analysis (VTA) [18] are efficient algorithms that conservatively resolves call sites without any help from points-to analysis. Grove et al. [8] 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 [21] 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. [14] 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 [11]. Most context-sensitive points-to analysis algorithms (e.g., [12, 17, 16, 19]) 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.

6 Conclusion

In this paper we have proposed Type Flow Analysis (TFA), an algorithm that constructs call graph edges for Object-Oriented programming languages. Different from points-to based analysis, we do not require a heap abstraction, so the computation is purely relational. We have proved that in the context-insensitive setting, our result is equivalent to that would be produced by a subset-based points-to analysis, regarding the core Object-Oriented language features. We have implemented the algorithm in the Soot compiler framework, and have conducted preliminary evaluation by comparing our results with those produced by the built-in CHA and points-to analysis algorithms in Soot on a selection of 13 benchmark programs from SPECjvm2008 and DaCapo benchmark suites, and achieved promising results. In the future we plan to develop context-sensitive analysis algorithms based on TFA.

References

1. Soot. `https://sable.github.io/soot/`. Accessed: 2019-06-10.
2. Specjvm2008. `https://www.spec.org/jvm2008/`. Accessed: 2019-06-18.
3. L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
4. David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, 1996.
5. Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java

benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, 2006.

6. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, 1995.
7. Mary F. Fernández. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 103–115, 1995.
8. David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, 1997.
9. P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43—-68, 1990.
10. George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, 2013.
11. Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 153–169, 2003.
12. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.
13. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16:973—-989, 1987.
14. Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 474–486, 2016.
15. Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis, 1991.
16. Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 17–30, 2011.
17. Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, 2006.
18. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 264–280, 2000.
19. Tian Tan, Yue Li, and Jingling Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium, SAS'16*, 2016.
20. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146—-160, 1972.

21. Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, 2000.