A Relational Static Semantics for Call Graph Resolution

June 18, 2019

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. In this paper, we propose a new approach called type flow analysis that represent the 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 variables, 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 the result of which a subsequent optimization may reduce the cost of virtual function calls or perform 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) [**?**,**?**] and Rapid Type Analysis (RTA) [**?**] and Variable Type Analysis (VTA) [**?**], conservatively assign each variable a set of class definitions, with relatively low precision. Alternatively, with the help of an abstract heap, one may take advantage of points-to analyses [**?**] 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 purpose of the conservative analyses, such as CHA, RTA and VTA, is to provide an efficient way to resolve calling edges, which usually executes linear-time in the size of a program, by focusing on the types that are collected at the receiver. For instance, let x be a receiver with declared class A, then at a call site $x.m()$, then 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 the nearest parent class that defines $m()$ will also be included. For x a receiver with declared interface I, then 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

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

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

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

Fig. 2. VTA facts on the example

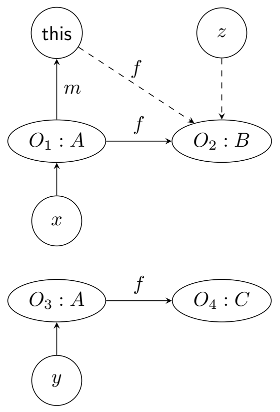

Fig. 3. Points-to results on the example

call site $x.m()$ 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 from RTA. VTA defines a node for each variable, method, method parameter and field. Class names are treated in a similar way as to variables, and propagation of value 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 give $\{A, B, C\}$ as the set of types reaching $A.f$.) 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 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 $O_1.m.this$ to O_2 is shown in dashed arrow. By return

$$\dfrac{\dfrac{\dfrac{z = x.m(), \textsf{[call]}}{A.m.this.f \sqsubseteq A.m.return} \qquad \dfrac{\dfrac{z = x.m(), \textsf{[call]}}{x \sqsubseteq A.m.this} \quad \dfrac{x.f = b, \textsf{[store]}}{x \xrightarrow{f} b}}{A.m.this \xrightarrow{f} b}, \textsf{[load]}}{b \sqsubseteq A.m.return} \qquad \dfrac{z = x.m(), \textsf{[call]}}{A.m.return \sqsubseteq z}}{b \sqsubseteq z}$$

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

of the method call $z = x.m()$, variable z receives O_2 of type B from $O_1.m.this.f$, which gives the most precise type 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. However, points-to analysis often requires to construct a heap abstraction, which brings in extra information especially in the case when we are only interested in the type 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 $\mathtt{VAR}$, 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 filed accesses. This provides us more ways to enlarge the relations $\sqsubseteq$ and $\rightarrow$. For example given $x \xrightarrow{f} y$ and $x \sqsubseteq z$, we can derive $z \xrightarrow{f} y$.
Taking the example from Figure 1, we are able to collect the store relation $x \xrightarrow{f} b$ from line 5. Since $x \sqsubseteq C.m.this$, we derive $C.m.this \xrightarrow{f} b$, which together with $C.m.this.f \sqsubseteq z$, further derives $b \sqsubseteq z$. 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 similar way.
We have proved that in the context insensitive inter-procedural analysis, the TFA is as precise as subset based points-to analysis regarding type related information. Since points-to analysis can be enhanced with context-sensitivity on both variables and objects (e.g., [**?**,**?**,**?**]), our type flow analysis only requires to consider context on variables, which is left for future work. The context-insensitive type flow analysis has been implemented in SOOT [**?**], and has been tested on a collection of benchmark programs from specjvm2008 and DaCapo. (Need more citations here, and brief the experimental results.)

2 Type Flow Analysis

We define a core calculus in Figure 5 consisting of most of the key Object-Oriented language features. 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. Two auxiliary functions are given. Function *fields* maps class names to its fields, and *class* provides types (or class names) for objects. Given class c, if $f \in fields(c)$, then $type(c, f)$ is the defined class type of field f in c. Similarly, give an object o, if $f \in fields(class(o))$, then $o.f$ may refer to an object of type $type(class(o), f)$, or an object of any of its subclass at runtime.

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

Fig. 5. Abstract syntax for the core language.

3 Optimization

4 Experiment

5 Related Work