Practical Virtual Method Call Resolution in Java [5]

Vijay Sundaresan, Chrislain Razafimahefa,
Raja Vallée-Rai, and Laurie Hendren

November 15, 1999

This paper intends to resolve method call in Java by means of the following flow-insensitive techniques:

- reaching-type analyses
  - variable-type analysis
  - declared-type analysis
- points-to analysis
  - refers-to analysis

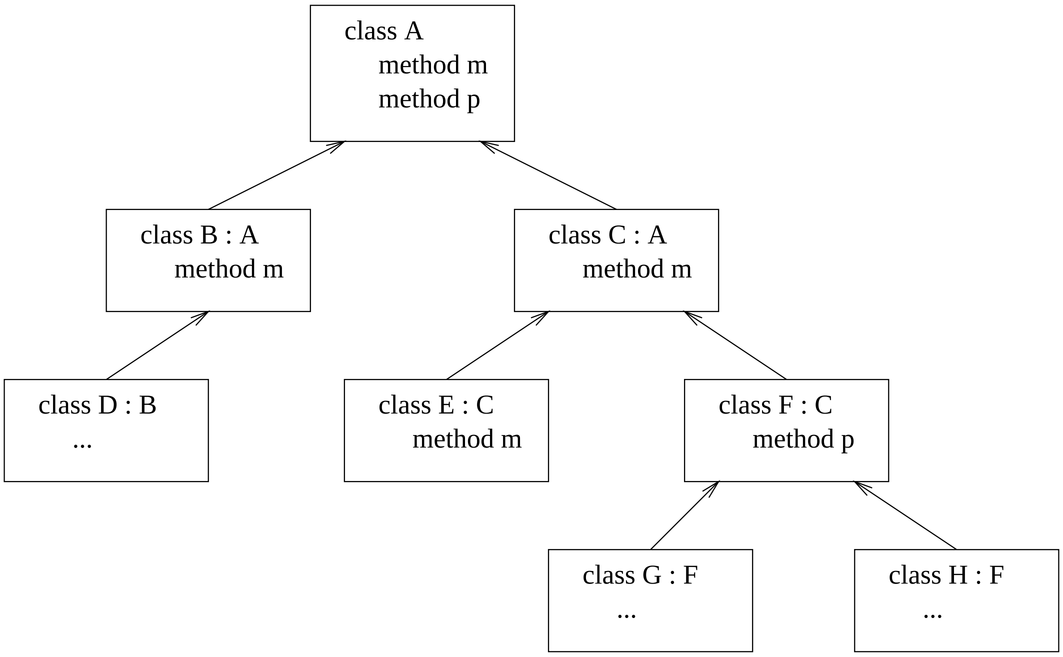
Implementations are made possible by use of the Soot framework and a specific type propagation graph.

Compared to a standard class hierarchy analysis, reaching-type analyses leads to call graphs with up to 44% fewer edges and up to 48% fewer nodes.

Class Hierarchy Analysis [1]

It aims at statically determining what methods can be invoked by calls:

- to reduce their cost
- to detect sites for method inlining
- to provide an accurate call grpah



For instance, if the method `p` in `F` sends the message `m` to `self`...

Sketched Implementation

Static class analysis earlier focused on the *singleton* class and the *union* class sets. Class hierarchy analysis introduces the *cone* class set.

Focusing on singleton class is not agressive enough: it misses 50% of messages statically bound using class hiearchy analysis.

One strategy can be...

Each method can be associated with an *applies-to* set:

$$\begin{array}{ll} A :: m & \{A\} \\ B :: m & \{B, D\} \\ C :: m & \{C, F, G, H\} \\ E :: m & \{E\} \end{array}$$

At a message send, we take the class set inferred for the receiver:

`o.m();` $\{F, G, H\}$

If only one method's applies-to set overlaps the receiver's class set, then that is the only method that can be invoked and the message send can be replaced with a direct procedure call.

Class Hierarchy Analysis for Java

We want to determine at compile-time a call graph with as few nodes and edges as possible.

Hierarchy definition

Given a receiver o with a declared type d, the set of possible run-time types of o, $hierarchy\_types(o, d)$, is defined a follows:

1. If o has a declared type C, $hierarchy\_types(o, C)$ includes C plus all subclasses of C.
2. If o has a declared interface type I, *hierarchy_types(o,I)* includes:
   (a) the set of all classes that implement I or implement a subinterface of I, call this set *implements(I)*, plus
   (b) all subclasses of *implements(I)*.

Building the call graph

- One node per reachable method.
- Each node contains a collection of *call sites*.
- Edges go from call sites to nodes, whenever a call is possible, using an approximation of the run-time types of the receiver.

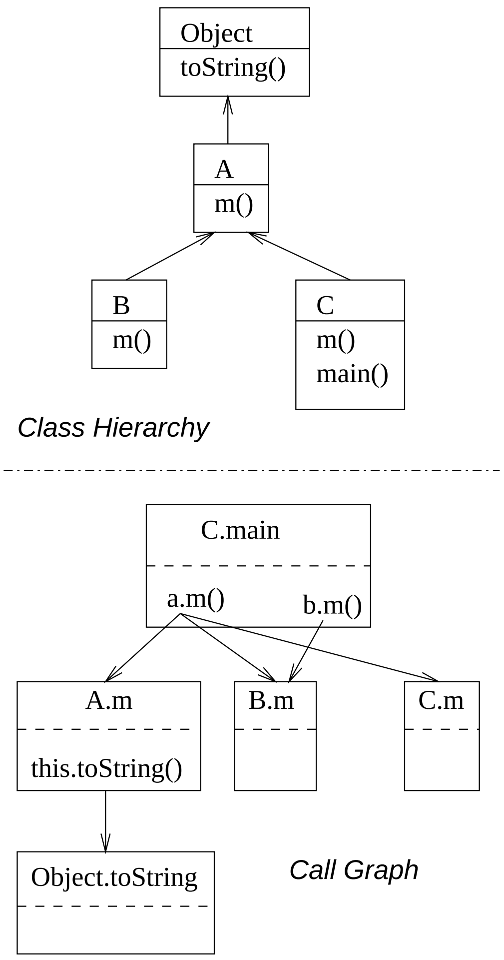
Example of a Call Graph

```
class A extends Object {
    String m() {
        return(this.toString());
    }
}

class B extends A {
    String m() { ... }
}

class C extends A {
    String m() { ... }
    public static void main(...) {
        A a = new A();
        B b = new B();
        String s;
        ...
        s = a.m();
        s = b.m();
    }
}
```

Source Code



- Such a call graph may contain spurious edges (due to conservative approximations) and nodes (due to spurious edges).
- It may be built iteratively using a worklist strategy.
- Here, $hierarchy\_types$ is the estimate $runtime\_types$.

Rapid Type Analysis

A receiver can only have a type of an object that has been instantiated with a `new`.

Use of rapid type analysis leads to a lighter call graph and therefore to further optimizations.

Rapid type definition

Given a receiver o with declared types C w.r.t. program P:

$$rapid\_types(C, P) = hierarchy\_types(C) \cap instantiated\_types(P)$$

The Soot Framework [2]

Soot is a framework for developing new optimizations for Java bytecode which consists of:

1. A set of intermediate representations (Baf, Jimple, Grimp).
2. A set of classes that manipulate these IRs (generating the IRs, building flows, analysing flows).

A complete Java to Java bytecode compiler is also being developed (see `http://www.sable.mcgill.ca/soot`).

Why new IRs?

Optimizing the Java bytecode directly is awkward since:

- Expressions are not explicit (parsing is necessary).
- The stack implicitly participates in each computation.
- Expressions are not as clear as 3-address code.
- There are two sorts of variables (explicit and implicit).
- Stack locations are untyped.

The Jimple Representation [4]

Jimple is a typed 3-address code representation of bytecode and is a more adapted form for performing optimizations.

```
public int stepPoly(int x)
{
    if (x < 0)
    {
        System.out.println("foo");
        return -1;
    }
    else if (x <= 5)
        return x * 5;
    else
        return x * 5 + 16;
}
```

Java

```
public int 'stepPoly'(int)
{
    Test r0;
    int i0, $i1, $i2, $i3;
    java.io.PrintStream $r1;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    $r1 = java.lang.System.out;
    $r1.println("foo");
    return -1;

label0:
    if i0 > 5 goto label1;

    $i1 = i0 * i0;
    return $i1;

label1:
    $i2 = i0 * 5;
    $i3 = $i2 + 16;
    return $i3;
}
```

Jimple

- The stack is eliminated by means of local variables,
- then, copy propagation and renaming are performed
- before typing the code and eventually reducing the number of locals.

Reaching-type Analyses

PROPERTY 1 (REACHING-TYPE PROPERTY)
For a type A to reach a receiver o, there must be some execution path through the program which starts with a call of a a constructor of the form $v = new\ A()$ followed by a chain of assignments of the form $x_1 = v, x_2 = x_1, \ldots, x_n = x_{n-1}, o = x_n$.

Both new analyses proceed by:

1. building the *type propagation graph*,
2. initializing the graph with type information generated by *new* statements, and
3. propagating type information along directed edges.

The *runtime_types* approximation will be built this way:

- A receiver o is associated with a node *representative*(o).
- A node n is associated with a set of types *reaching_types*(n).
- The types reaching o is the set *reaching_types*(*repreentative*(o))

Variable Type Analysis

Variable type analysis uses the 'name' of a variable as its representative.

We first create nodes

Given a program P, nodes are creating as follows:

- for every class C
  for every object type field f
  $\rightarrow$ create a node labeled with f
- for every method $C.m$
  $\rightarrow$ create a node labeled $C.m.this$
  $\rightarrow$ create a node labeled $C.m.return$
  for every object type formal parameter p_i of $C.m$
  $\rightarrow$ create a node labeled $C.m.p_i$
  for every object type local variable l_i of $C.m$
  $\rightarrow$ create a node labeled $C.m.l_i$

Then we create edges

- For each assignment statement of the form $lhs = rhs$
  $\rightarrow$ add an edge from *representative(rhs)* to *representative(lhs)*

- For each $C'.m'$ that is a target of a method call of the form $lhs = o.m(a_1, \ldots, a_n)$; or $o.m(a_1, \ldots, a_n)$; in the conservative call graph
  $\rightarrow$ add an edge from $representative(o)$ to $C'.m'.this$
  if the return type is not void
  $\rightarrow$ add an edge from $C'.m'.return$ to the representative for lhs
  for each object type argument a_i
  $\rightarrow$ add an edge from the representative of a_i to the representative of the matching parameter of $C'.m'$

And propagate types

- We first visit each $lhs{=}new A()$ statement and add type A to the set of representative of lhs.
- We collapse each strongly-connected component into one supernode along with the corresponding set of types.
- Types are propagated in a single pass in a breadth-first manner in the resulting DAG.

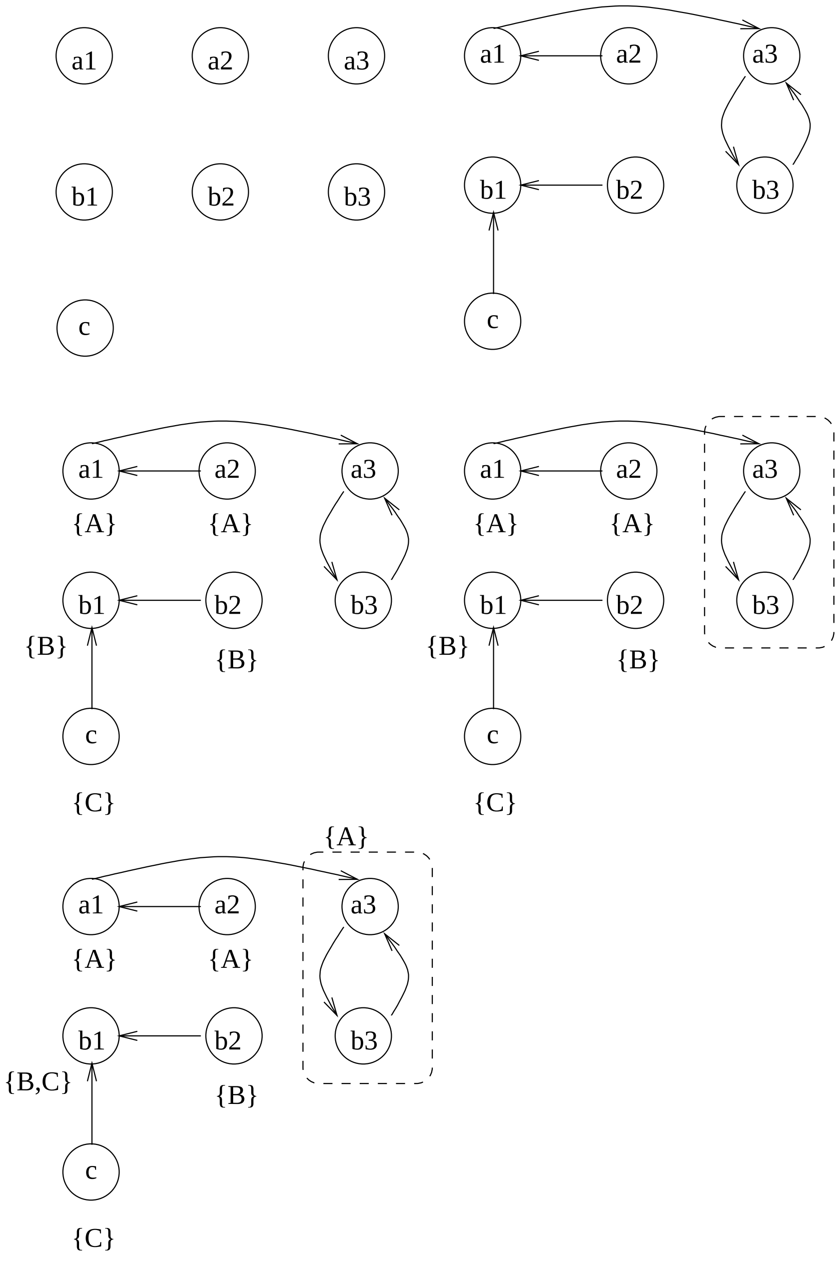
The type propagation graph includes at most $2M + P + L + F$ nodes (linear size (?)).
The number of edges is slightly more difficult to estimate (number of classes $\times$ number of method calls (?)).

Example

```
A a1, a2, a3;
B b1, b2;
C c;

a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```



Declared Type Analysis

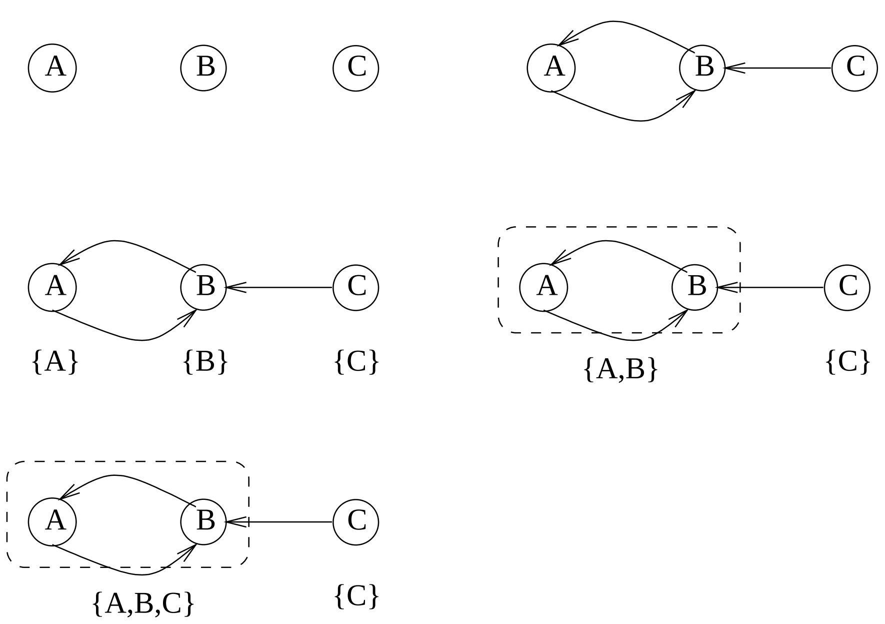
Declared type analysis uses the declared type of the variable as its representative (instead of the variable name).

Basically, it puts all variables with the same declared type into the same equivalence class.

```
A a1, a2, a3;
B b1, b2;
C c;

a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```



The size is considerably smaller, but the final answer is not as precise.

Refers-to Analyses

This is a simple version of Steensgaard's points-to algorithm [3] modified to work with the Jimple representation.

- *Reference nodes* represent Java variable references.
- *Abstract location nodes* represent heap locations.
- *New_reference nodes* are reference nodes created at object instantiation sites.

The analysis works as follows:

- Each reference refers-to a unique abstract location.
- Each assignment $lhs = rhs$ causes the abstract locations for lhs to be merged with the abstract locations for rhs
- such that eventually, each reference refers to an *equivalence class representative*.

If a receiver o refers-to the same ECR as a new reference node with type A, we conclude that o might have a runtime type of A.

Example of Refers-to Analysis

For the previous example, we sould have two classes:

{`newA.1, newA.2, a1, a2, a3, b3`}
{`newB.1, newB.2, newC.1, b1, b2, c`}

where `newA.1` represents the first allocation site with type `A`.

- This is more precise than declared-type analysis (it finds that all of the `a` variables and `b3` can only refer-to objects of type `A`).
- Yet it is less precise than variable-type analysis (it conservatively grouped `c` with objects of type `B`).

Call Graph built using Class Hierarchy Analysis

9 benchmarks (2 from the McGill set and 7 from the SPECjvm suite).

	virtual				interface			
Name	1	2	> 2	total	1	2	> 2	total
puzzle	1489	26	109	1629	0	1	9	11
jimple	6217	51	554	6837	17	163	693	1079
mtrt	3486	55	172	3720	91	26	40	171
jess	4188	66	187	4457	112	26	45	198
compress	2548	55	166	2776	91	26	40	171
db	2710	55	169	2943	91	26	60	191
mpegaudio	2961	65	191	3224	106	26	40	186
jack	3258	377	183	3930	107	26	90	252
javac	5639	219	764	6732	92	26	102	244

Many `invokevirtual/interface` call sites have edges to only one callee method.

Improvements over the Conservative Call Graph

Name	Analysis	Nodes removed	Edges removed	Call Edge Reductions $\rightarrow 0$	$\rightarrow 1$	$\rightarrow 2$
mtrt	rapid-type	35%	19%	435	42	50
	refers-to	35%	19.2%	497	29	50
	declared-type	39%	23%	498	48	43
	variable-type	44%	28%	670	56	48
mpegaudio	rapid-type	34.8%	20.8%	432	51	31
	refers-to	35.4%	21.3%	479	38	31
	declared-type	39.7%	25.6%	459	55	56
	variable-type	43.8%	31.7%	668	65	61

- RTA removes between 9% to about 35% of the total number of methods.
- Refers-to analysis performs slightly better than RTA in removing dead methods.
- Declared-type analysis shows an increase of 15% to 20% in this number.
- Variable-type analysis shows a similar improvement over declared-type analysis.

References

[1] David Grove Jeffrey Dean and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *LNCS 952 (ECOOP)*, 1995.

[2] Etienne Gagnon Laurie Hendren Patrick Lam Raja Vallée-Rai, Phong Co and Vijay Sundaresan. Soot: a java bytecode optimization framework. In *http://www.sable.mcgill.ca/publications/#cascon99*, 1999.

[3] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96*, 1996.

[4] Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for anlyses and transformations. In *http://www.sable.mcgill.ca/publications/#report4*, 1998.

[5] Raja Vallée-Rai Vijay Sundaresan, Chrislain Razafimahefa and Laurie Hendren. Practical virtual method call resolution for java. In *?*, 1998.