Practical Virtual Method Call Resolution for Java

Sable TR 1999-2

Introduction

- **Objective**

  To determine at compile time a call graph with as few nodes and edges as possible

- **Background**

  .Soot Framework and Jimple

- **Techniques**

  . Class Hierarchy Analysis

  . Rapid Type Analysis

  . Reaching type Analysis

  --- Variable-type Analysis

  --- Declared-type Analysis

- **Comparison**

Background

- Jimple(stackless) versus Java Bytecode(stack-based)[Why manipulate Jimple instead of Java Bytecode?]

Disadvantages of Java bytecode:

1) Expressions are not explicit
2) Expressions can be arbitrary large
3) Concrete expressions can not always be constructed
4) Simple transformations become complicated

Background

- Example of Jimple

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

(a) java source

```
public int stepPoly(int)
 { java.io.PrintStream r1;
    Example r0;
    int i0, i1, i2, i3;
    r0 := @this;
 i0 =@parameter0;
 if i0 >= 0 goto label10;
 r1 =java.lang.System.out;
 r1.println("error");
 return -1;
 label 0: if i0 > 5 goto label1;
i1 = i0 * i0;
return i1;
 label 1: i3=i0*5;
  i2 = i3 +16;
return i2; }
```

(b) Jimple representation

Background

- Some Notes on Jimple

  1) Translated from Java bytecode instead of high- level Java programs

  2) Features:

  . Relatively few kinds of statements

  . Operands are either typed variable references or constants

  . Identity statements

Class Hierarchy Analysis

- Class Hierarchy Analysis

--- A standard method for conservatively estimating the run-time types of receivers .

--- Definition of *hierarchy_types(o,d)*

Given a receiver *o* of a declared type *d*, *hierarchy_types(o,d)* for Java is defined as follows:

. If *d* is a class type C,
hierarchy_types(o,d) includes C plus all subclasses of C

. If *d* is a interface type I,
hierarchy_types(o,d) includes: 1) the set of all classes that implement I or implement a sub-interface of I, which we call implements(I), plus 2) all subclasses of implements(I).

Class Hierarchy Analysis

- Call Graph

  --- Nodes

  . A node represents a method that can be reached starting at any entry point. (For a single-threaded non-applet program, only one entry point: the *main* method.)

  . A node contains a collection of call sites. Consider a method M from class C with n method calls in its body. M is represented by a node labeled C.M, which contain entries for each call site, which we denote C.M[c_1] to C.M[c_n].

Class Hierarchy Analysis

- Call Graph

  --- Edges

  . Edges go from call sites within a call graph node, to call graph nodes;

  . Edges represents possible calling relationship between call sites and nodes

  . How to add calling edges from a virtual method or interface call? (Note the potential runtime types for the receiver is a set, call this set *runtime_types(o)*)

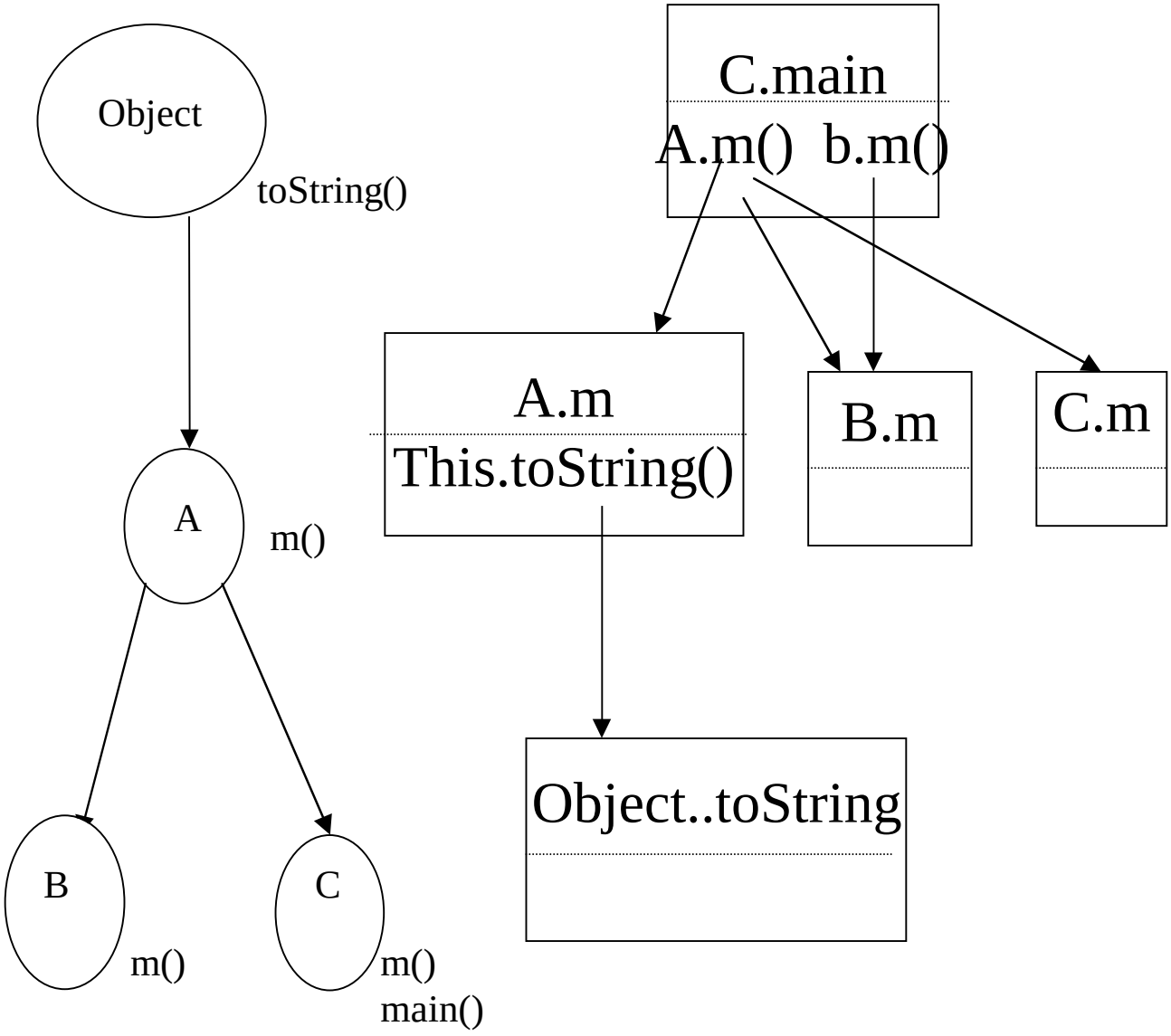
Class Hierarchy Analysis

- An Example

```
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();  }
       }
```

Class Hierarchy Analysis

- An Example(Continued)



a) Class Hierarchy

b) Call Graph

Class Hierarchy Analysis

- Building the Conservative Call Graph

--- Using *hierarchy_types* as the estimate for *runtime_types* for determining the edges from virtual method call sites

--- Worklist strategy

Starting with nodes for all possible entry points(e.g. main, start, run). As each node (method) is added to the call graph, edges from the call sites in the node are also added. If the target of an edge is not already in the call graph, then it is added to the call graph and to the worklist.

Class Hierarchy Analysis

- Could we do better?

Objective

To determine at compile time a call graph with as few nodes and edges as possible

Observation

1) Spurious types may lead to spurious edges

2) Spurious nodes are included when all incoming edges to the node are spurious

Principle for Solutions

To provide better approximations of the runtime types of receivers. (One old and two new solutions are given later …)

Measurement

To concentrate on the number of call edges instead of the accuracy of the receiver type resolution.

Rapid Type Analysis

- Observation

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

- Solution

  --- Collect the set of object types instantiated in the program P, call this instantiated_types(P).

  --- Given a receiver o with declared type C with respect to program P, define rapid_types(C,P) = hierarchy_types(o, C) and Instantiated_types(P)

  ---- Use rapid_types to estimate runtime_types

Rapid-type Analysis

- Coarse Reaching Type Analysis

  A type A reaches a receiver o if there is an instantiation of an object of A(i.e. new A()) anywhere in the program, and A is a plausible type for o using hierarchy analysis.
- We can have fine-grain Reaching Type Analysis ...

Reaching-type Analysis

- Observation

  Assuming no alias relation between variables, 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 constructor of the form v=new A() followed by some chain of assignments of the form x_1=v,x_2=x_1,…x_{n-1}=x_n,o=xn

Reaching-type Analysis

- Framework

1) Building a type propagation graph
2) Initializing the graph with type information generated by new() statements
3) Propagating type information along directed edges

- Terminology

1) Representative(o): each receiver o is associated with some node in the type propagation graph, called representative(o)
2) Reaching_types(n): after type propagation each node n is associated with a set of types, called reaching_types(n)

Reaching_type Analysis

- Variable-type Analysis

  Let the name of the receiver o be the representative of o. In Jimple, there are three kinds of variable references:

  1) Ordinary references: of form a, and refer to locals and parameters. The name C.m.a is used as the representative

  2) Field references: of form a.f where a could be a local, a parameter, or this. The name C.f where C is the name of the class defining field f is used as the representative

  3) Array references: of form a[x] where a is a local or parameter, and x is a local, parameter, or constant. The name C.m.a is used as the representative, similar to the ordinary reference case.

Variable-type Analysis

- Constructing the Type Propagation Graph

  --- Nodes

  For every class C included in P,

  1)for every field f in C, where f has an object(reference) type, create a node labeled with C.f

  For every method C.m that is included in the conservative call graph of P

  1)for every formal parameter p_i of C.m, where p_i has an object type, create a node labeled C.m.p_i

  2) for every local variable l_i of C.m, where l_i has an object type, create a node labeled C.m.l_i

  3) create a node labeled C.m.this to represent the implicit first parameter

  4) create a node labeled C.m.return to represent the return value C.m

Variable-type Analysis

- Constructing the Type Propagation Graph

  ---Edge

  Assignment Statements of form lhs=rhs(lhs and rhs are ordinary, field or array reference):

  Add a directed edge from the representative node for rhs to the representative node of lhs.

  Method Calls of form lhs = o.m(a1,a2,…,an); or o.m(a1,a2,…,an);. O must be a local, a parameter, or the special identifier this. The arguments must be a constant, a local, or parameter name.

  (To be continued)

Variable_type Analysis

- Constructing the Type Propagation Graph

---Edge

Method Calls (continued)

The method call corresponds to some call site, call itC.m[I], in the conservative call graph.

For each C'.m' that is the target of of C.m[I] in conservative call graph**,**

1) Add an edge from the representative of o to C'.m'.this
2) if the return type is not void add an edge from C'.m'.return to the representative for lhs
3) for each argument a_i that has object type, add an edge from the representative of $\mathbf{a_i}$ to the representative of the matching parameter of C'.m'.

Variable-type Analysis

- Alias

  The assignment rules assume that if a and b are alias, then they should correspond to the same node in the graph.

--- Ordinary references

// No Problem

Locals and parameters cannot be aliased in Java

--- Field references

// No problem

All instances of objects with that field are represented as one node in the graph

Variable-type Analysis

- Alias

--- Array reference

/*problem: Several different variable names may refer to the same array.

e.g. A[] a = new A[10]; Object o1 = a; Object o2 = o1; A[] b = (A[]) o2; */

Solution: when adding edges for assignments of the form lhs = rhs, where both sides are of type java.lang.Object, or when at least one side has an array type, edges are added in both directions between the representative of rhs and lhs.

Variable-type Analysis

- Type propagation graph

--- Size

Nodes: <= 2M+P+L+F

M: the number of methods

P: the total number of parameters

L: the total number of locals

F: the number of fields

Edges: $O(C*M_c)$

Assignment statements: At most One edge for each Assignment statement

Method calls: the number of edges depends on the number of targets for call sites. Worst Case: $O(C*M_c)$ where C is the number of classes and M_c is the number of method calls.

Variable-type Analysis

- Type propagation graph

--- Initialization

For each statement of the form lhs = new A(); or lhs = new A[n] , add type A to the Reaching types set of representative node for lhs.

--- propagation

Phase 1: Collapse strongly-connected components

Phase 2: Propagation on the DAG

(single pass, breadth-first)

Complexity: Both detecting Strongly-connected components and propagation have O(max(N,E)) operations, the most expensive of which is a union of two ReachingType sets.

Declared-Type Analysis

- Declared-Type Analysis

  --- Similar to variable-type analysis,

  --- the declared type of the variable instead of the variable name is used as the representative.

  Thus, basically it is like putting all variables with the same declared type into the same equivalent class

  --- faster but more imprecise

Experimental Results

- Benchmarks
- Improvements over the Conservative Call Graph
- Conclusion