

A Relational Static Semantics for Call Graph Construction

Xilong Zhuo¹ and Chenyi Zhang²

¹College of Information Science and Technology, Jinan University, China

²College of Information Science and Technology, Jinan University, China

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.

Keywords: Type Analysis; Static Analysis; Method Resolving; Call Graph

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...

2. Type Flow Analysis

We define a core calculus consisting of most of the key object-oriented language features...

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

2.1. $c \dashrightarrow y$

2.2. $x \sqsubseteq y$

2.3. $x \xrightarrow{f} y$

3. Implementation

The analysis algorithm is written in Java, and is implemented in the Soot framework...

3.1. Static Analysis Tool

3.2. Dynamic Profiler

Since the benchmarks do not provide the run-time type of method receiver, we implement a dynamic profiling tool to record types which a receiver can access at run-time. To achieve this, we instrument statements into the target benchmark. After this instrumentation, the run-time type will be extracted during the benchmark execution. We consider this output as groundtruth and compare it with our static result in section 4.2. There are four manners to instrument the source code to record the run-time type of a method receiver.

3.2.1. Insert First

In this manner, the type-recorded statements will be insert before the first statement of a method block and the type of “this” reference in that method will be recorded. The reason we only have to record “this” reference is that a receiver is always passed into “this” reference in a method, except for static methods.

3.2.2. Insert Before

Statements will be inserted before invocations and the type of receiver will be recorded in this manner. It is more straightforward than the method we discuss about recording “this” reference.

3.2.3. Insert Last

This manner is similar with “Insert First”, except that statements are inserted after the last statement of a method block. We also record “this” reference in this way.

3.2.4. Insert After

Statments will be inserted right after invocations and the type of receiver will be recorded. It is similar with “Insert Before”, except that statements are inserted at different position. We use this manner in our implementation and the reason will be discuss in section 3.2.5

3.2.5. Our Instrumentation Manner

We use “Insert After” as our instrumentation manner. The reason is mainly due to the Java specification of constructor that the first statement in constructor should be either another constructor of its own or its super class. Therefore, we will get JVM violation error if we instrument a statement before the first statement in construtor. We illustrate this example on 1. So both “Insert First” and “Insert Before” manners can not be applied under this circumstance. We choose “Insert After” over “Insert Last” for reason that it’s more straightforward. The code before and after instrumentation are shown in 2 and 3, respectively.

4. Evaluation

We evaluate our approach by measuring its performance on 13 benchmark programs...

```

1  class A {
2      public A() {
3          //insert statements here will violate JVM specification
4          super();    //invoke super class constructor
5      }
6      public A(int i) {
7          //insert statements here will violate JVM specification
8          this();    //invovke another constructor of its own
9      }
10 }

```

Fig. 1. Java specification on constructor

```

1  class A {
2      public void m1() {
3          B b = new B();
4          b.m2();    //invocation here
5      }
6  }
7  class B {
8      public void m2() {
9          ...
10     }
11 }

```

Fig. 2. Example code before instrumentation

4.1. Efficiency

We executed each benchmark program 10 times with the CHA, PTA and TFA algorithms. We calculated the average time consumption...

4.2. Precision

4.2.1. Reflection Call

Reflection in Java programming language is a advanced feature which provides ability to inspect and manipulate a Java class at runtime. It brings in extra complexity on programs and the behaviour is hard to predict statically. In 4 we pick some codes using reflection in the benchmark programs to discuss how reflection works and what is our treatment on that. Note that we reorganize and simplify the real code a little to concentrate on the main point of reflection usage. We discuss in three cases:

```

1  class A {
2      public void m1() {
3          ...
4          b.m2();    //invocation here
5          String record = RecordUtils.id(InvokeExpression, b);
6          RecordUtils.record(record);
7      }
8  }

```

Fig. 3. Example code after instrumentation

```

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 }

```

Fig. 4. Example code of reflection

- Object Creation

Related code ranges from line 6 to 10. A new object of type "SPECJVMBenchmarkBase" will be created by invoking the method "newInstance()" on variable *c* at line 10. *c* is an object of "Constructor" type and it refers to a specific constructor of class "SPECJVMBenchmarkBase". Our method discard the type information of new object in these case because it's difficult to statically identify which constructor will be invoked. *e.g.* At line 6, If statement *Class.forName()* receive argument from outside liked user input or loading file with content of class name, then we could not find out the real type of *benchmarkClass*. As a result, the run time type of *c* and *benchmark* could not be identified neither.

- 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 the method named "invoke()" of that object. This effect is just like a normal invocation at line 12. The invocation receiver is passed to the first argument of method "invoke()". If this invocation is static, a **null** reference will be passed to the first argument. We do not consider these effects of method invocation in reflection manner for the same reason we discuss about object creation.

- Field Modification

The way to change a field using reflection is similar to processing method invocation. The last two line illustrate changing value of a field named "f" on object "benchmark" into a new object. We discard this effect as well because of the difficulty analyzing on which class holds the target field.

4.2.2. Java Native Interface Call

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 5. 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

```
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 }
```

Fig. 5. Example code of JNI

actually implemented in different program languages. 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 that invocation “*a.m()*” at line 8 will be discarded.

4.2.3. Library

4.2.4. Array Approximation

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.

6. Conclusion

In this paper we have proposed Type Flow Analysis (TFA), an algorithm that constructs call graph edges for Object-Oriented programming languages.