Under consideration for publication in Formal Aspects of Computing

A Relational Static Semantics for Call Graph Construction

Xilong Zhuo[1] and Chenyi Zhang[2]

[1]College of Information Science and Technology, Jinan University, China
[2]College 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. $\mathbf{c} \dashrightarrow \mathbf{y}$

2.2. $\mathbf{x} \sqsubseteq \mathbf{y}$

2.3. $\mathbf{x} \xrightarrow{f} \mathbf{y}$

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 voilation 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...

```
class A {
  public A() {
    //insert statements here will violate JVM specification
    super();     //invoke super class constructor
  }
  public A(int i) {
    //insert statements here will violate JVM specification
    this();     //inovke another constructor of its own
  }
}
```

Fig. 1. Java specification on constructor

```
class A {
  public void m1() {
    B b = new B();
    b.m2();     //invocation here
  }
}
class B {
  public void m2() {
    ...
  }
}
```

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...

```
class A {
  public void m1() {
    ...
    b.m2();     //invocation here
    String record = RecordUtils.id(invokeExpression, b);
    RecordUtils.record(record);
  }
}
```

Fig. 3. Example code after instrumentation

4.2. Precision

4.2.1. Reflection Call

4.2.2. JNI Call

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.