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, the most popular static analysis framework for Java. We use jimple as our intermediate representation. We do not take common types (*e.g.* , int and float) under our consideration. That are irrelevant to our analysis. We keep method invocation from Java advanced features liked reflection or JNI as unresolvable. More detail will be discussed in 4.2.1 and 4.2.2. Conservative approximation is performed on invocation of methods from libraries (*e.g.* , JDK) and array accesses. We will describe these strategies in 4.2.3 and 4.2.4.

A Static analysis tool is implemented to process our algorithm and extract static result of type solution. In addition, we implement a dynamic profiler to record the run time type of variable, which can be used to compare with the static result. Detail of static analysis tool and dynamic profiler will be discuss in 3.1 and 3.2, respectively.

3.1. Static Analysis Tool

Our static analysis tool is implemented in Java and aims to analyze Java programs. It takes Java bytecode files as input. Any other format of Java code will be accepted if it can be translated to jimple representation by Soot(*e.g.* , jar files). The implementation of our static analysis tool can be splited into four step as follow.

- Code translation
  Target code is loaded by Soot and translated to IR of jimple format.
- Basic relation generation
  We iterate all statements on jimple IR to generate those basic relations based on the basic relation difinition.
- Fixpoint calculation
  After basic relations are generated, we use the extended relation rule to perform fixpoint calculation.
- Result extraction
  When the fixpoint is achieved, all set of reaching types of all variable are immutable. We extract those set of reaching types as our final result.

Fig 1 shows the whole process of our static analysis tool. Data input and generated are represented in dash arrow. The final result is generated in the “Result Extraction” phase and represented in green background.

3.2. Dynamic Profiler

Since the benchmarks do not provide the grountruth of 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.

- 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.
- Insert Before

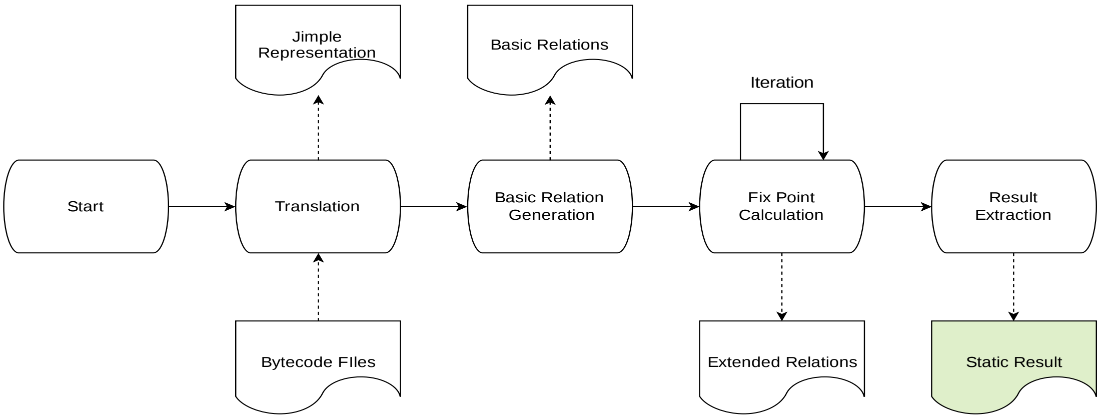

Fig. 1. Process of static analysis tool

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.

- 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.
- 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 take this manner in our implementation and the reason will be discuss in section 3.2.1

3.2.1. Our Instrumentation Manner

We take "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 Listing 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 Listing 2 and Listing 3, respectively. For Listing 3, at line 5, the funtion $RecordUtils.id()$ takes an invocation expression ($invokeExprssion$) and a method receiver (b) as parameters and return an unique representation string for this invocation. For this example, the unique representation is "A:m1:b:B:m2:4". This means b will call method $m2$ of class B, at line 4 in method $m1$ of class A. It also shows that b is of type B at this position.

```
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
  }
}
```

Listing 1: Java specification on constructor

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

Listing 2: Example code before instrumentation

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

Listing 3: Example code after instrumentation

4. Evaluation

We evaluate our approach by measuring its performance on SPECjvm2008, which contains 12 benchmark programs in total. We conduct all of our experiments on a laptop equipped with an Intel i5-8250U CPU at 1.60 GHz and 8 GB memory, running Ubuntu 16.04LTS with OpenJDK 1.8.0.

We compare our approach against the default implementation of Class Hierarchy Analysis (CHA) and context-insensitive points-to analysis that are implemented by Soot team. The reason we do not compare against Variable Type Analysis (VTA) due to its unavaliable implementation. The only implementation for Java we can find is also implemented by Soot team, but it is embeded as a subprocess to optimize points-to analysis. Under this circumstance, we compare our approach against VTA in precision with manual analysis in section 1. Implementation of VTA will be left for our future work. During the evaluation the following tow research questions are addressed.

- **RQ1** How efficient is our approach compared with the traditional class hierarchy analysis and points-to analysis?
- **RQ2** How accurate is our type resolution when comparing with the other analysis?

We answer RQ1 and RQ2 in section 4.1 and section 4.2, respetively.

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 Listing 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:

- Object Creation
  Related codes range from line 6 to 10. A new obejct 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 on analyzing which class holds the target field.

```
 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  }
```

Listing 4: Example code of reflection

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 Listing 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 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.

```
 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 }
```

Listing 5: Example code of JNI

4.2.3. Library

Library are those codes included in the application and used to accomplish specific function(*e.g.* , JDK library, three-party library). Listing 6 shows a common case of JDK invocation. We do not analyze the detail logic inside library code which are written at line 10-11. Instead, we perform an over approximation on library invocation, based on the method difinition which appears at line 9. We assume that library invocation will return the difinition type and any subtype of this difinition type as the result type. For Listing 6, $sb2$ will receive $\{StringBuilder,\ any\_subtype\_of\_StringBuilder\}$ as the set of reaching types under this over approximation strategy.

```
 1 import java.lang.StringBuilder;
 2
 3 public void m() {
 4   StringBuilder sb = new StringBuilder();
 5   StringBuilder sb2 = sb.append("abc"); <——
 6 }
 7
 8 //   @Override
 9 //   public StringBuilder append(String str) {
10 //      ...
11 //      ...
12 //   }
```

Listing 6: Example of JDK library call

4.2.4. Array Approximation

We perform a conservative treatment on array accesses that all type information that flows to one member of an array flows to all members of that array. Codes in Listing 7 are used to explain how this approximation works. Type information of A and B are flow to the first member and the second member of array arr, respectively. We approximate that these two type information flow to array arr. Loadding an element of array will receive all types that array can access(*e.g.* , b will receive $\{A, B\}$ as the set of reaching types, which is the same set of types that array arr can access).

```
public void m() {
  Object[] arr = new Object[2]{};
  arr[1] = new A();
  arr[2] = new B();
  Object b = arr[1];  <——
}
```

Listing 7: Example of array access

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.