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. c⇢ y

2.2. x⊑ y

2.3. x$\xrightarrow{f}$ 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

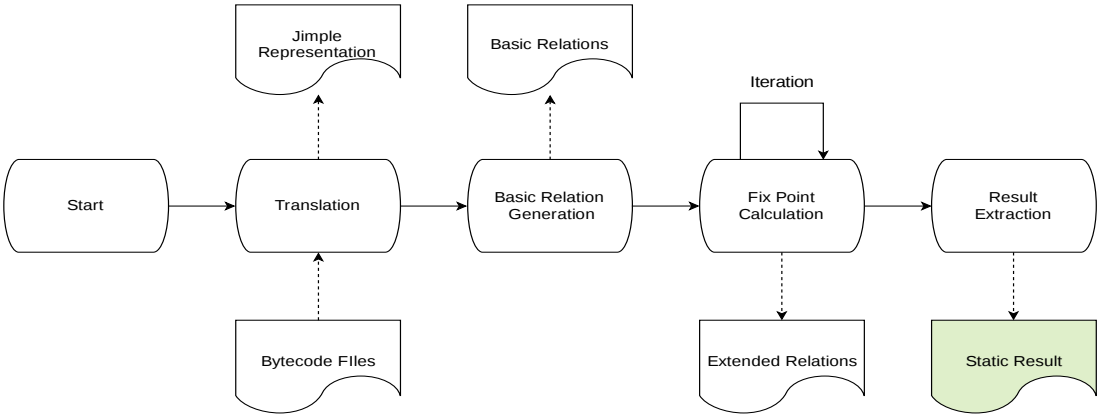

Fig. 1. Process of static analysis tool

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.

```
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();    //inovke another constructor of its own
9    }
10 }
```

Listing 1: Java specification on constructor

```
1  class A {
```

```
2   public void m1() {
3     B b = new B();
4     b.m2();     //invocation here
5   }
6 }
```

Listing 2: Example code before instrumentation

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

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 avaliable implementation for Java 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. The choice of the context-insensitive points-to analysis is due to our approach also being context-insensitive, thus the results will be comparable. We use iteration algorithm for points-to analysis to calculate a fixpoint for the same reason. In the following tables we use CHA, PTA and TFA to refer to the results related to class hierarchy analysis, points-to analysis and type flow analysis, respectively. 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 algorithm when comparing with the other analysis?
- **RQ3** Does our approach spend less storage comparing with PTA since we do not have to maintain an abstract heap?

We answer these three questions in section 4.1, section 4.2 and section 4.3, respetively.

4.1. Efficiency

We executed each benchmark program 10 times with the CHA, PTA and TFA algorithms. We calculated the average time consumption (in seconds) as displayed at column $\mathbf{T}_{CHA}$(s), $\mathbf{T}_{PTA}$(s) and $\mathbf{T}_{TFA}$(s) of the Table in Fig. 2. We counted the sizes of each generated realtion (*i.e.* , the type flow realtion '$\dashrightarrow$', variable partial order '$\sqsubseteq$' and the field access relation '$\longrightarrow$'). It provides an estimation of size for the problem we are treating. The result shows that our approach consums more time than CHA, for the reason that CHA do not have to analyze detail logic inside the program but only analyze the class and interface hierarchical structure. TFA is in general more efficient than points-to analysis. The runtime cost in TFA basically depends on the size of generated relations, as well as the relational complexity as most of the time is consumed to calculate a fixpoint. For PTA it also require extra time for maintaining and updating a heap abstraction.

Benchmark	$\mathbf{T_{CHA}}(s)$	$\mathbf{T_{PTA}}(s)$	$\mathbf{T_{TFA}}(s)$	$\mathbf{R}_{\dashrightarrow}$	$\mathbf{R}_{\sqsubseteq}$	$\mathbf{R}_{\rightarrow}$	$\mathbf{R}_{total}$
check	5.33	71.98	10.46	6847	15599	10089	32535
compiler	5.76	72.01	11.48	6665	14898	10433	31996
compress	5.72	71.06	12.19	6410	14344	10322	31076
crypto	5.60	69.25	9.21	6459	14424	10152	31035
derby	5.83	70.57	11.65	6887	14853	10603	32343
helloworld	6.26	70.72	13.10	6149	13652	10071	29872
mpegaudio	6.29	70.45	10.65	6197	13737	10084	30018
scimark	6.77	71.37	11.24	6366	14678	10214	31258
serial	7.06	70.41	10.98	6627	14309	10341	31277
startup	5.62	69.09	10.73	6239	13723	10094	30056
sunflow	5.58	72.93	11.12	6167	13675	10077	29919
xml	6.55	139.08	13.84	6866	16039	11060	33965

Fig. 2. Runtime cost with different analysis

[1] $\mathbf{R}_{relation}$ denotes different type of relation we generated.

4.2. Accuracy

We answer the seconde question about accuracy in two indexes: precision and recall. Their difinitions are given in equation 1 and 2, respectively.

$$precision = M/S \tag{1}$$

$$recall = M/D \tag{2}$$

"D" stands for the number of dynamic record, "S" refers to the numer of static record, and "M" represents the number of matching record.

A sound algorithm should catch all type information in run time, *i.e.* , the recall should be 100%. Precision indicates how accurate the generated set of types is. In general, a more acurrate algorithm often generate a smaller set of types for each calling variable. We calculate recall and precision of different approach, as displayed in Fig. 3

The result shows that in general our approach can achieve higher precision than CHA and closer precision than PTA. The imprecision is mainly due to our approximation on library invocation and array, which we will discuss in section 4.2.3 and section 4.2.4. Besides, the code coverage will issue imprecision since SPECjvm provides common codes for all benchmarks and some of them would not be excuted when a specific benchmark is excuting, *i.e.* , these codes will be analyzed statically but not generate any record in run time.

For soundness, the recall of both TFA and PTA can not achieve to 1.00 in most benchmarks except *helloworld*. We analyzed those missed records and find out the reason is due to some Java advance features, liked reflection call and JNI call, which we will describe in section 4.2.1 and section 4.2.2. In addition, callback mechanism will raise unsoundness under our treatment on reflection, JNI and library call. We will discuss it in section 4.2.5. The result shows that PTA produces lower recall than TFA in general. This difference is due to the reachability analysis used on PTA implemented by Soot team. Method invoked by reflection, or invoked as a callback function inside reflection, JNI and library, will be denoted as unreachable under this analysis. As a consequence, some methods would not be analyzed by PTA. Note that even the unreachable method would not be analyzed, a variable receiving its type from JDK would be assigned to possible types based on the preprocessing from Soot.

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:

Benchmark	D	$\mathbf{M}_{CHA}$	$\mathbf{M}_{PTA}$	$\mathbf{M}_{TFA}$	$\mathbf{R}_{CHA}$	$\mathbf{R}_{PTA}$	$\mathbf{R}_{TFA}$	$\mathbf{P}_{CHA}$	$\mathbf{P}_{PTA}$	$\mathbf{P}_{TFA}$
check	150	150	145	145	1.00	0.97	0.97	0.07	0.19	0.18
compiler	515	515	463	506	1.00	0.90	0.98	0.28	0.81	0.70
compress	353	353	330	345	1.00	0.93	0.98	0.19	0.49	0.46
crypto	454	454	393	407	1.00	0.87	0.90	0.25	0.70	0.65
derby	639	639	603	632	1.00	0.94	0.99	0.35	0.97	0.88
helloworld	21	21	21	21	1.00	1.00	1.00	0.01	0.03	0.03
mpegaudio	255	255	241	248	1.00	0.95	0.97	0.15	0.40	0.37
scimark	386	386	357	362	1.00	0.92	0.94	0.20	0.54	0.50
serial	458	458	354	410	1.00	0.77	0.90	0.18	0.67	0.62
startup	1429	1429	1358	1360	1.00	0.95	0.95	0.82	2.23	2.07
sunflow	226	226	208	217	1.00	0.92	0.96	0.13	0.36	0.33
xml	521	521	436	504	1.00	0.84	0.97	0.27	0.81	0.65

Fig. 3. Accuracy with different analysis

[1] **D** denotes dynamic records.
[2] $\mathbf{M}_{algorithm}$, $\mathbf{R}_{algorithm}$, $\mathbf{P}_{algorithm}$ denote matching count, recall and precision of different algorithms, respectively.

- 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, *sb*2 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
```

```
//  public StringBuilder append(String str) {
//    ...
//    ...
//  }
```

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

4.2.5. Callback

A callback function is known as a "call-after" function. It is widely used in programs. By passing a argument to one invocation as the receiver which is expected to trigger a callback invocation. In functional language a callback funtion can be passed as argument directly. Listing 8 shows the callback mechanism. The function *callback*() will be invoked after the invocation of *trigger*(). Since we do not analyze the effect of reflection call, JNI call, and library call, we are unable to catch the type flow information inside *a.trigger*(*b*) if it happens to be one of these calls (*e.g.* , *trigger*() is a method of a library class). Note that we only conservatively approximate on the return type of a library invocation, but not analyze the detail logic inside a library method.

```
class A {
    public void trigger(B b) {
        b.callback();  // here the callback function actually executes
    }
}

class B {
    public void callback() {...}
    public void static main() {
        A a = new A();
        B b = new B();
        a.trigger(b);
    }
}
```

Listing 8: Callback mechanism

4.3. Space

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.