Soot: a framework for analysis and optimization of Java

www.sable.mcgill.ca

Java .class files

- Contain fields, methods and attributes
- Fields: instance variables or class variables
- Methods: contain Java bytecode

```
// java source
int cc (int x, int y) {
    int z;
    z = x*y;
    return z; }
```

```
// bytecode(javap –c)
Method int cc (int, int) {
    0 iload 1
    1 iload 2
    2 imul
    3 istore 3
    4 iload 3
    5 ireturn  }
```

.jimple files

- An Intermediate Representation

```
// java source
int cc (int x, int y) {
    int z;
    z = x*y;
    return z; }
```

```
// bytecode(javap –c)
Method int cc (int, int) {
    0 iload 1
    1 iload 2
    2 imul
    3 istore 3
    4 iload 3
    5 ireturn  }
```

```
// jimple(java soot.Main -f jimple)
int cc(int, int)
  {  int i0, i1, i2;
     i0 := @parameter0: int;
     i1 := @parameter1: int;
     i2 = i0 * i1;
     return i2;
  }
```


Intermediate Representations

- Bytecode vs. 3-address code

Bytecode:

- Each instruction has implicit effect on stack
- No types for local variables
- > 200 kinds of insts

Typed 3-address code:

- Each stmt acts explicitly on named variables
- Types for each local variable
- Only 15 kinds of stmt

Do analysis on JIMPLE 3-address code IR.

Intermediate Representations

- Source vs. 3-address code

Source

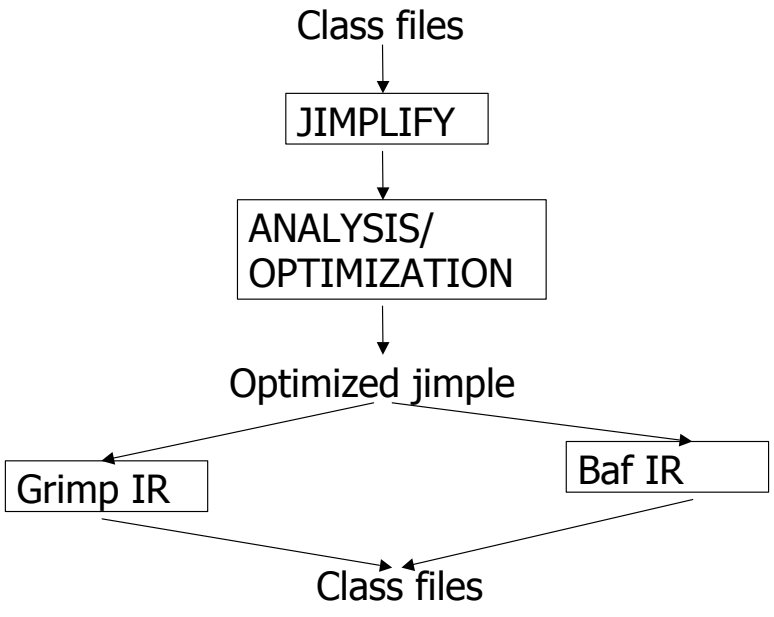
- Irregular structure (somewhat)
- Complex statements and expressions

3-address code:

- More regular structure
- 15 kinds of stmts, simple expressions and statements

Analysis is simpler and more effective on JIMPLE 3-address code than source!

Overview of Soot



Advantages of Jimple and Soot

- JIMPLE
  - Typed local variables
  - Simple expressions (1 operator / stmt)
- SOOT
  - Uses and defs are easily available
  - Soot provides data-flow analysis framework
  - Hierarchy information available
  - Can construct call graph

Understanding Jimple

- Run soot: java soot.Main –f jimple MyClass

```
public class A {
    main(String[] args) {
        A a = new A();
        a.m();
    }
    public void m() {
    }
}
```

```
public class A extends java.lang.Object
  {
    public void <init>() {
      A r0;
      r0 := @this: A;
      specialinvoke r0.
      <java.lang.Object: void <init>()>();
      return; }
    …
```

Understanding Jimple, cont.

```
public class A {
   main(String[] args) {
      A a = new A();
      a.m();
   }
   public void m() {
   }
}
```

```
…
public void m()
   {
      A r0;
      r0 := @this: A;
      return;
   }
…
```


Understanding Jimple, cont.

```
public class A {
   main(String[] args) {
      A a = new A();
      a.m();
   }
   public void m() {
   }
}
```

```
…
main(java.lang.String[]) {
      java.lang.String[] r0;
      A $r1, r2;
      r0 := @parameter0: java.lang.String[];
      $r1 = new A;
      specialinvoke $r1.<A: void <init>()>();
      r2 = $r1;
      virtualinvoke r2.<A: void m()>();
      return; }
 }
```

Phase in Soot

- In SOOT, each phase is implemented by a *Pack. Each pack is a collection of transformers, each corresponding to a subphase.*
- Phase *cg*
- The Call Graph Constructor computes a call graph for whole program analysis. The different phases in this pack are different ways to construct the call graph. Exactly one phase in this pack must be enabled.
- *cg.spark*----spark is a flexible points-to analysis framework (http://www.sable.mcgill.ca/publications/thesis/#olhotakMastersThesis)
- Phase *wjtp*
- Whole Jimple Transformation Pack
- Run after Phase *cg*

An Example To Get Call Graph

```
public class YourMain
{   public static void main(String[] args){
        if(args.length == 0) {
            System.exit(0);
        }
    PackManager.v().getPack("wjtp").
        add(new Transform("wjtp.name",
                    YourTransformer.v()));
    soot.Main.main(args);
  }
}
```

```
public class YourTransformer
        extends SceneTransformer{

        ... ...
         protected void internalTransform ( String
        phaseName, Map options){
            CallGraph cg
                =Scene.v().getCallGraph();

            ......
        }
}
```

To Get a Call Graph Generated by Points-to Analysis

- To run the program: java YourMain --app -p cg.spark on-fly-cg:true -w TargetJavaApplication
- -app : application mode, processing all possible reachable classes
- -w: whole program mode
- More Soot command line options please refer to
  http://www.sable.mcgill.ca/soot/tutorial/usage/

CallGraph in Soot

- CallGraph is made up of Edges
- Edge(MethodOrMethodContext src, Stmt srcUnit, MethodOrMethodContext tgt)
- From the call site (Stmt), you should figure out the possible targets in CHA.

Jimple Grammar-Statement

stmt ⟶ *assignStmt* | *identityStmt* | *gotoStmt* | *ifStmt* | *invokeStmt* | *switchStmt* | *monitorStmt* | *returnStmt* | *throwStmt* | *breakpointStmt* | *nopStmt*;
assignStmt ⟶ local = *rvalue*; | field = *imm*; | local . field = *imm*; | local [*imm*] = *imm*;
identityStmt ⟶ local := @this: type; | local := @parameter*n*: type; | local := @exception;
gotoStmt ⟶ `goto` label;
ifStmt ⟶ `if` *conditionExpr* `goto` label;
invokeStmt ⟶ `invoke` *invokeExpr*;

Resources

- Master Thesis,"SOOT: A Java Bytecode Optimization Framework",by Raja Vallee-Rai
  http://www.sable.mcgill.ca/publications/thesis/#korMastersThesis
- Tutorial: http://www.sable.mcgill.ca/soot/tutorial/
- paul: /grad/cs515/soot222/soot-2.2.2/tutorial