Soot, a Tool for Analyzing and Transforming Java Bytecode

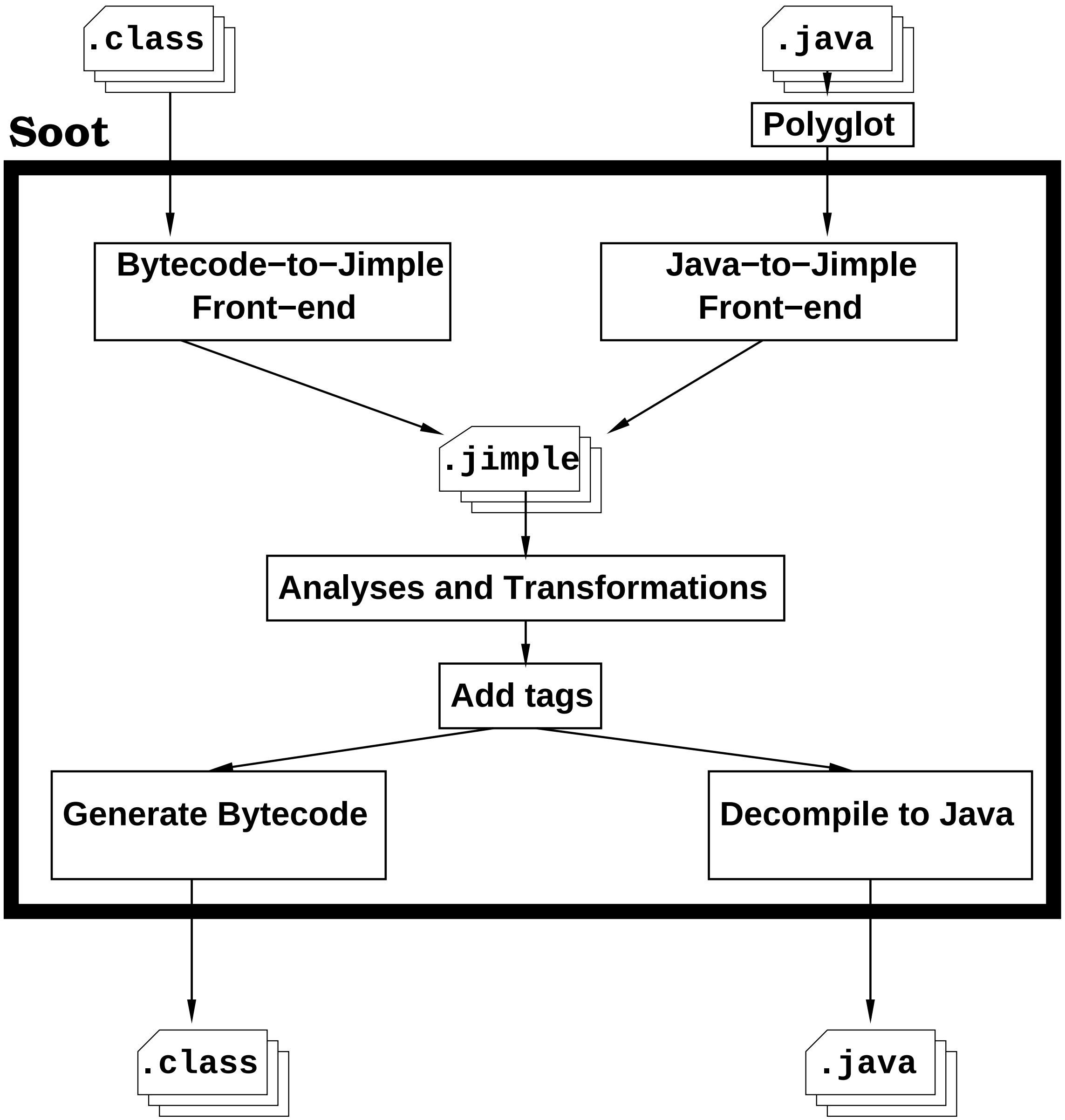
presenter: Ondřej Lhoták, McGill University

Soot toolkit

Soot provides:

- Convenient IRs (mainly Jimple)
- Existing analyses and transformations
- Framework for new analyses, transformations, code generation
- Dava decompiler
- Eclipse plugin for visualization
- Whole-program analysis framework

Soot Overview



Jimple

Jimple is:

- principal Soot Intermediate Representation
- 3-address code in a *control-flow graph*
- a *typed* intermediate representation
- *stackless*

Jimple example

Java:

```
public int bar(int a, int b) {
    return a+b;
}
```

Jimple:

```
public int bar(int, int) {
    Foo this;
    int a, b, $i0;

    this := @this;
    a := @parameter0;
    b := @parameter1;
    $i0 = a + b;
    return $i0;
}
```

Converting bytecode → Jimple → bytecode

- These transformations are relatively hard to design so that they produce correct, useful and efficient code.
- Worth the price, we do want a 3-addr typed IR.

raw bytecode	typed 3-address code (Jimple)
○ each inst has implicit effect on stack	○ each stmt acts explicitly on named variables
○ no types for local variables	○ types for each local variable
○ $>$ 200 kinds of insts	○ only 15 kinds of stmts

Bytecode → Jimple

- Naive translation from bytecode to untyped Jimple, using variables for stack locations.
- splits DU-UD webs (so many different uses of the stack do not interfere)
- types locals (SAS 2000)
- cleans up Jimple

Java → Jimple

- Input: Polyglot AST generated from .java sources
- Compile AST to Jimple
- Generate Jimple methods/classes for implicit Java features (initializers, inner class accessor methods, class literals, assertions)
- Output: Jimple to be analyzed/optimized, eventually converted to bytecode
- Combination of Polyglot, Java-to-Jimple, Jimple-to-Bytecode passes forms a complete Java compiler equivalent to javac.

This part is unchanged in abc.

Jimple → Bytecode

- A naive translation introduces many spurious stores and loads.
- Two approaches (CC 2000),
  - aggregate expressions and then generate stack code; or
  - perform store-load and store-load-load elimination on the naive stack code.

Weaving example – source

```
public class Foo {
    public int foo(int x, int y, int z) {
       return bar(x, y, z);
    }
    public int bar(int a, int b, int c) {
        return a+b+c;
    }
}
aspect A {
    before(Foo x) :
    call(int bar(int,int,int)) && target(x) {
        System.out.println(x);
    }
}
```

Weaving example – original bytecode

```
public int foo(int x, int y, int z)
0:    aload_0
1:    iload_1
2:    iload_2
3:    iload_3
4:    invokevirtual     Foo.bar (III)I (7)
7:    ireturn
```

Weaving example – weaving by hand

```
public int foo(int x, int y, int z)
0:     invokestatic      A.aspectOf ()LA; (14)
3:     aload_0
4:     invokevirtual     A.before$0 (LFoo;)V (20)
7:     aload_0
8:     iload_1
9:     iload_2
10:    iload_3
11:    invokevirtual     Foo.bar (III)I (9)
14:    ireturn
```

Weaving example – ajc weaving

```
public int foo(int x, int y, int z)
0:     aload_0
1:     iload_1
2:     iload_2
3:     iload_3
4:     istore            %4
6:     istore            %5
8:     istore            %6
10:    astore            %7
12:    invokestatic      A.aspectOf ()LA; (52)
15:    aload             %7
17:    invokevirtual     A.ajc$before$A$124 (LFoo;)V (56)
20:    aload             %7
22:    iload             %6
24:    iload             %5
26:    iload             %4
28:    invokevirtual     Foo.bar (III)I (37)
31:    ireturn
```

Weaving example – original Jimple

```
public int foo(int, int, int)
{
    Foo this;
    int x, y, z, $i0;

    this := @this;
    x := @parameter0;
    y := @parameter1;
    z := @parameter2;
    $i0 = this.bar(x, y, z);
    return $i0;
}
```

Weaving example – woven Jimple

```
public int foo(int, int, int)
{
    Foo this;
    int x, y, z, $i0;
    A theAspect;

    this := @this;
    x := @parameter0;
    y := @parameter1;
    z := @parameter2;
    theAspect = A.aspectOf();
    theAspect.before$0(this);
    $i0 = this.bar(x, y, z);
    return $i0;
}
```

Weaving example – bytecode from Jimple

```
public int foo(int x, int y, int z)
0:    invokestatic      A.aspectOf ()LA; (14)
3:    aload_0
4:    invokevirtual     A.before$0 (LFoo;)V (20)
7:    aload_0
8:    iload_1
9:    iload_2
10:   iload_3
11:   invokevirtual     Foo.bar (III)I (9)
14:   ireturn
```

Intraprocedural analyses and transformations

- local packer (“register allocation” on bytecode locals)
- copy propagation
- constant propagation
- common subexpression elimination
- partial redundancy elimination
- dead assignment elimination
- unreachable code elimination
- branch simplification

Law of Demeter benchmark

In method lawOfDemeter.objectform.Pertarget.fieldIdentity:

- ajc: 616 locals
- ajc+Soot: 3 locals
- abc: 3 locals

Law of Demeter benchmark

In method lawOfDemeter.objectform.Pertarget.fieldIdentity:

ajc:	616 locals	45.9 seconds
ajc+Soot:	3 locals	14.1 seconds
abc:	3 locals	1.0 second

Adding analyses and transformations

Soot provides tools:

- control flow graphs
- def/use relationships
- fixed-point flow analysis framework
- method inliner

These are useful to have available for:

- weaving itself
- optimizing woven code

Dava decompiler

```
public int foo(int x, int y, int z)
{
    A.aspectOf().before$0(this);
    return this.bar(x, y, z);
}
```

- Dava decompiles bytecode with strange aspect-generated control flow that breaks other decompilers.
- Dava is integrated with Soot and abc.
  We could produce annotated decompiled output (e.g. comments showing pointcuts).

Eclipse plugin

- Soot can be run as a plugin from Eclipse.
- Soot includes a tagging framework to communicate analysis information to Eclipse for visualization. (CC2004, eTX2004)
- Could be used to communicate aspect-specific information.

Whole-program analyses

- CHA call graph
- VTA – more precise call graph (OOPSLA2000)
- Spark: context-ins. points-to and call graph (CC2003)
- Paddle: BDD based framework for context-sensitive:
  - points-to analysis
  - call graph analysis
  - cflow analysis
  - type analysis (`instanceof` checks)
  - side-effect analysis (aspect purity)
  - escape analysis (`thisJoinPoint[StaticPart]`)