Analyzing Java Programs with Soot

Bruno Dufour

based on material by:
Eric Bodden, Laurie Hendren, Patrick Lam, Jennifer Lhoták, Ondřej Lhoták and Feng Qian
McGill University

http://www.sable.mcgill.ca/soot/
What is Soot?

- a free compiler infrastructure, written in Java (LGPL)
- was originally designed to analyze and transform Java bytecode
- original motivation was to provide a common infrastructure with which researchers could compare analyses (points-to analyses)
- has been extended to include decompilation and visualization
Soot has many potential applications:

- used as a stand-alone tool (command line or Eclipse plugin)
- extended to include new IRs, analyses, transformations and visualizations
- as the basis of building new special-purpose tools
Soot: Past and Present

- Started in 1996-97 with the development of *coffi* by Clark Verbrugge and some first prototypes of *Jimple IR* by Clark and Raja Vallée-Rai.

- First publicly-available versions of Soot 1.x were associated with Raja’s M.Sc. thesis

- New contributions and releases have been added by many graduate students at McGill and research results have been the topics of papers and theses.

www.sable.mcgill.ca/publications/
Soot 1.x has been used by many research groups for a wide variety of applications. Has also been used in several compiler courses. Last version was 1.2.5.

Soot 2.0 and the first version of the Eclipse Plugin were released - June 2003 - JIT for PLDI 2003.

Soot 2.3.0: Java 5 support
Soot 2.4.0: partial Reflection support
This tutorial is based on Soot 2.4.0.
Soot Overview

- Java source
- SML source
- Scheme source
- Eiffel source

- javac
- MLJ
- KAWA
- SmallEiffel

- class files

- SOOT
  - Produce Jimple 3-address IR
  - Analyze, Optimize and Tag
  - Generate Bytecode

- Optimized class files + attributes

- Eclipse
  - Interpreter
  - JIT
  - Adaptive Engine
  - Ahead-of-Time Compiler
**Soot IRs**

**Baf:** is a compact rep. of *Bytecode* (stack-based)

**Jimple:** is Java’s *simple*, typed, 3-addr (stackless) representation

**Shimple:** is a *SSA*-version of **Jimple**

**Grimp:** is like **Jimple**, but with expressions *aggregated*

**Dava:** structured representation used for Decompiling Java
Jimple is:

- principal Soot Intermediate Representation
- 3-address code in a control-flow graph
- a typed intermediate representation
- stackless
- special variables for this and parameters
- only simple statements, never nested
Kinds of Jimple Stmts I

- Core statements:
  - NopStmt
  - DefinitionStmt: IdentityStmt, AssignStmt

- Intraprocedural control-flow:
  - IfStmt
  - GotoStmt
  - TableSwitchStmt, LookupSwitchStmt

- Interprocedural control-flow:
  - InvokeStmt
  - ReturnStmt, ReturnVoidStmt
Kinds of JimpleStmts II

- **ThrowStmt**
  throws an exception

- **RetStmt**
  not used; returns from a JSR

- **MonitorStmt**: EnterMonitorStmt, ExitMonitorStmt
  mutual exclusion
this.m();

Where’s the definition of this?

IdentityStmt:

- Used for assigning parameter values and this ref to locals.
- Gives each local at least one definition point.

Jimple rep of IdentityStmts:

```
  r0 := @this;
  i1 := @parameter0;
```
public int foo(java.lang.String) { // locals
    r0 := @this; // IdentityStmt
    r1 := @parameter0;

    if r1 != null goto label0; // IfStmt

    $i0 = r1.length(); // AssignStmt
    r1.toUpperCase(); // InvokeStmt
    return $i0; // ReturnStmt

label0: // created by Printer
    return 2;
}
Converting bytecode $\rightarrow$ Jimple $\rightarrow$ 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.

<table>
<thead>
<tr>
<th>raw bytecode</th>
<th>typed 3-address code (Jimple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>○ each inst has implicit effect on stack</td>
<td>○ each stmt acts explicitly on named variables</td>
</tr>
<tr>
<td>○ no types for local variables</td>
<td>○ types for each local variable</td>
</tr>
<tr>
<td>○ $&gt;200$ kinds of insts</td>
<td>○ only 15 kinds of stmts</td>
</tr>
</tbody>
</table>
Soot builds data structures to represent:
- a complete environment (Scene)
- classes (SootClass)
- Fields and Methods (SootMethod, SootField)
- bodies of Methods (come in different flavours, corresponding to different IR levels, ie. JimpleBody)

These data structures are implemented using OO techniques, and designed to be easy to use and generic where possible.
Body-centric View

SootMethod

JimpleBody

Chain

getActiveBody()

getLocals()

getUnits()

getTraps()

Chain

Chain
Getting a UnitGraph

- UnitGraph
  - getUnits()
  - new BriefUnitGraph()

- SootMethod
  - getBody()
  - getActiveBody()

- JimpleBody
  - getLocals()
  - getUnites()
  - getTraps()

- Chain
  - Chain
  - Chain
What to do with a UnitGraph

- `getBody()`
- `getHeads()`, `getTails()`
- `getPredsOf(u)`, `getSuccsOf(u)`
- `getExtendedBasicBlockPathBetween(from, to)`
Control-flow units

We create an OO hierarchy of units, allowing generic programming using Units.

- **Unit**: abstract interface
- **Inst**: Baf’s bytecode-level unit
  
  \[
  \text{(load } x)\]

- **Stmt**: Jimple’s three-address code units
  
  \[
  (z = x + y)\]

- **Stmt**: also used in Grimp
  
  \[
  (z = x + y \ast 2 \% n;)\]
Soot Philosophy on Units

Accesses should be **abstract** whenever possible!

Accessing data:

- `getUseBoxes()`, `getDefBoxes()`, `getUseAndDefBoxes()`

(also control-flow information:)

- `fallsThrough()`, `branches()`, `getBoxesPointingToThis()`, `addBoxesPointingToThis()`, `removeBoxesPointingToThis()`, `redirectJumpsToThisTo()`
What is a Box?

\[ s: \ x = \ y \ op \ z \]

AssignStmt

AssignStmt

x OpExpr

y z

VB

VB

OpExpr

VB

VB

y

z
What is a DefBox?

List defBoxes = ut.getDefBoxes();

- method `ut.getDefBoxes()` returns a list of ValueBoxes, corresponding to all Values which get defined in `ut`, a Unit.

- non-empty for `IdentityStmt` and `AssignStmt`.

```
    ut:  x = [y op z];
```

`getDefBoxes(ut) = {x}`

(List containing a ValueBox containing a Local)
Value value = defBox.getValue();

- **getValue()**: Dereferencing a pointer.
  \[
  \text{x} \rightarrow x
  \]
- **setValue()**: mutates the value in the Box.
Opposite of defBoxes.
List useBoxes = ut.getUseBoxes();

- method ut.getUseBoxes() returns a list of ValueBoxes, corresponding to all Values which get used in ut, a Unit.
- non-empty for most Soot Units.

\[
\text{ut: } \quad x = y \text{ op } z;
\]

getUseBoxes(ut) = \{y, z, [y op z]\}  
(List containing 3 ValueBoxes, 2 containing Locals & 1 Expr)
Why Boxes?

Change all instances of \( y \) to 1:

Before:

```
AssignStmt
  VB
  OpExpr
    VB
    VB
      VB
      VB
        Y
        z
      setValue()  // ??
```

After:

```
AssignStmt
  VB
  OpExpr
    VB
    VB
      VB
      VB
        x
        OpExpr
          y
          z
```

AssignStmt
setValue()  // ??
/* Replace all uses of v1 in body with v2 */
void replace(Body body, Value v1, Value v2)
{
  for (Unit ut : body.getUnits())
  {
    for (ValueBox vb : ut.getUseBoxes())
      if (vb.getValue().equals(v1))
        vb.setValue(v2);
  }
}

replace(b, y, IntConstant.v(1));
Jimple provides the following additional accessors for special kinds of Values:

- `containsArrayRef()`, `getArrayRef()`, `getArrayRefBox()`
- `containsInvokeExpr()`, `getInvokeExpr()`, `getInvokeExprBox()`
- `containsFieldRef()`, `getFieldRef()`, `getFieldRefBox()`
Intraprocedural Outline

- About Soot’s Flow Analysis Framework
- Flow Analysis Examples
  - Live Variables
  - Branched Nullness
- Adding Analyses to Soot
Flow Analysis in Soot

- Flow analysis is key part of compiler framework
- Soot has easy-to-use framework for intraprocedural flow analysis
- Soot itself, and its flow analysis framework, are object-oriented.
Four Steps to Flow Analysis

1. Forward or backward? Branched or not?
2. Decide what you are approximating. What is the domain’s confluence operator?
3. Write equation for each kind of IR statement.
4. State the starting approximation.
HOWTO: Soot Flow Analysis

A checklist of your obligations:

1. Subclass *FlowAnalysis
2. Implement abstraction: merge(), copy()
3. Implement flow function flowThrough()
4. Implement initial values: newInitialFlow() and entryInitialFlow()
5. Implement constructor (it must call doAnalysis())
Soot provides you with:

- Impl of abstraction domains (flow sets)
  - Standard abstractions trivial to implement;
- An implemented flow analysis namely,
  - `doAnalysis()` method: executes intraprocedural analyses on a CFG using a worklist algorithm.
Soot Flow Analyses

AbstractFlowAnalysis

FlowAnalysis
  - Forward-PRE analy's
  - Backward-PRE analy's
  - Avail. Expr.
  - Array Bds

BranchedFlowAnalysis
  - Forward-
  - Casts
  - Nullness
  - Liveness
Backward vs. Forward Analyses

A forward analysis computes OUT from IN:

\[ s(i) \rightarrow f_s(i) \rightarrow f_t(f_s(i)) \]

A backward analysis computes IN from OUT:

\[ f_s(f_t(i)) \rightarrow f_t(i) \rightarrow f_t(i) \]
Outline: Soot Flow Analysis Examples

Will describe how to implement a flow analysis in Soot and present examples:

- live locals
- branched nullness testing
Example 1: Live Variables

A local variable \( v \) is **live** at \( s \) if there exists some statement \( s' \) using \( v \) and a control-flow path from \( s \) to \( s' \) free of definitions of \( v \).

\[
\begin{align*}
\text{a = z} & : \{z\} \\
\text{b = y} & : \{z, y\} \\
\text{x = y} & : \{z, y\} \\
\text{y = x} & : \{z, x\}, \{z, y\} \\
\text{z = 2} & : \{y\}, \{z, y\}
\end{align*}
\]
Steps to a Flow Analysis

As we’ve seen before:

1. Subclass *FlowAnalysis
2. Implement abstraction: merge(), copy()
3. Implement flow function flowThrough()
4. Implement initial values: newInitialFlow() and entryInitialFlow()
5. Implement constructor (it must call doAnalysis())
Step 1: Forward or Backward?

Live variables is a backward flow analysis, since flow $f^n$ computes IN sets from OUT sets.

In Soot, we subclass `BackwardFlowAnalysis`.

class LiveVariablesAnalysis extends BackwardFlowAnalysis<Unit,Set>
Step 2: Abstraction domain

Domain for Live Variables: sets of Locals
e.g. \{x, y, z\}

- Partial order is subset inclusion
- Merge operator is union

In Soot, we use the provided ArraySparseSet implementation of FlowSet.
Implementing an Abstraction

Need to implement `copy()`, `merge()` methods:

- `copy()` brings IN set to predecessor’s OUT set.
- `merge()` joins two IN sets to make an OUT set.
Signatures:

```java
void merge(Set src1, Set src2,
            Set dest);
void copy(Set src, Set dest);
```
Flow Sets and Soot

Soot provides special sets called FlowSets, which are often helpful.

**Impls:** ToppedSet, ArraySparseSet, ArrayPackedSet

```java
// c = a ∩ b
a.intersection(b, c);

// c = a ∪ b
a.union(b, c);

// d = c
// d = d ∪ {v}
c.complement(d);
d.add(v);
```
Digression: types of FlowSets

Which FlowSet do you want?

- **ArraySparseSet**: simple list
  
<table>
<thead>
<tr>
<th>foo</th>
<th>bar</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>bar</td>
<td>z</td>
</tr>
</tbody>
</table>

  (simplest possible)

- **ArrayPackedSet**: bitvector w/ map
  
<table>
<thead>
<tr>
<th>00100101</th>
<th>10101111</th>
<th>10000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100101</td>
<td>10101111</td>
<td>10000000</td>
</tr>
</tbody>
</table>

  (can complement, need universe)

- **ToppedSet**: 
  
  FlowSet & isTop()
  
  (adjoins a $\top$ to another FlowSet)
Step 2: \texttt{copy()} for live variables

protected void copy(Set src,  
                 Set dest) {
    dest.clear(); dest.addAll(src);
}
Step 2: `merge()` for live variables

In live variables, a variable \( v \) is live if there exists any path from \( d \) to \( p \), so we use `union`.

```java
void merge(...) {
    dest.clear();
    dest.addAll(src1Set);
    dest.addAll(src2Set);
}
```

Often, you may want to implement a more exotic merge.
Step 3: Flow equations

Goal: At a unit like \( x = y \times z \):
  - kill def \( x \);
  - gen uses \( y, z \).

How? Implement this method:

```java
protected void flowThrough
    (Set srcValue,
     Unit u,
     Set destValue)
```
Step 3: Copying

Need to copy \texttt{src} to \texttt{dest} to allow manipulation.

\begin{verbatim}
dest.clear();
dest.addAll(src);
\end{verbatim}
Step 3: Implementing flowThrough

Must decide what happens at each statement (in general, need to switch on unit type):

\[
\begin{align*}
\text{IN}[\text{ut}] &= \text{flowThrough}(\text{OUT}[\text{ut}]) \\
&= \text{OUT}[\text{ut}] \setminus \text{kills}[\text{ut}] \cup \text{gens}[\text{ut}]
\end{align*}
\]

flowThrough is the brains of a flow analysis.
Step 3: flowThrough for live locals

A local variable \( v \) is live at \( s \) if there exists some statement \( s' \) containing a use of \( v \), and a control-flow path from \( s \) to \( s' \) free of def’ns of \( v \).

Don’t care about the type of unit we’re analyzing: Soot provides abstract accessors to values used and defined in a unit.
Step 3: Implementing **flowThrough**: removing kills

```java
// Take out kill set:
// for each local v def’d in
// this unit, remove v from dest
for (ValueBox box : ut.getDefBoxes()) {
    Value value = box.getValue();
    if (value instanceof Local) {
        dest.remove(value);
    }
}
```
Step 3: Implementing **flowThrough**: adding gens

```java
// Add gen set
// for each local v used in
// this unit, add v to dest
for (ValueBox box : ut.getUseBoxes()) {
    Value value = box.getValue();
    if (value instanceof Local) {
        dest.add(value);
    }
}

N.B. our analysis is generic, not restricted to Jimple.
```
Step 4: Initial values

- Soundly initialize IN, OUT sets prior to analysis.
  - Create initial sets
    ```java
    Set newInitialFlow()
    {
        return new HashSet();
    }
    ```
  - Create initial sets for exit nodes
    ```java
    Set entryInitialFlow()
    {
        return new HashSet();
    }
    ```

Want conservative initial value at exit nodes, optimistic value at all other nodes.
Step 5: Implement constructor

```java
LiveVariablesAnalysis(UnitGraph g) {
    super(g);
    doAnalysis();
}
```

Causes the flow sets to be computed, using Soot’s flow analysis engine.

In other analyses, we precompute values.
You can instantiate an analysis and collect results:

```java
LiveVariablesAnalysis lv = new LiveVariablesAnalysis(g);
```

// return HashSets
// of live variables:
`lv.getFlowBefore(s);`
`lv.getFlowAfter(s);`

Enjoy: Flow Analysis Results
class VeryBusyExpressionAnalysis
    extends BackwardFlowAnalysis {
    [...]
}

Example 2: VeryB
public VeryBusyExpressionAnalysis(DirectedGraph g) {
    super(g);
    doAnalysis();
}

protected void merge(Object in1,
                     Object in2,
                     Object out) {
    FlowSet inSet1 = (FlowSet)in1,
                     inSet2 = (FlowSet)in2,
                     outSet = (FlowSet)out;
    inSet1.intersection(inSet2,
                        outSet);
}
protected void copy(Object source,
        Object dest) {
    FlowSet srcSet = (FlowSet) source,
        destSet = (FlowSet) dest;
    srcSet.copy(destSet);
}
protected void flowThrough(Object in,
          Object node,
          Object out) {

    FlowSet inSet = (FlowSet)source,
          outSet = (FlowSet)dest;

    Unit u = (Unit)node;

    kill(inSet, u, outSet);
    gen(outSet, u);
}
private void gen(FlowSet outSet, Unit u) {
    for (ValueBox useBox: u.getUseBoxes()) {
        if (useBox.getValue().instanceof BinopExpr) {
            outSet.add(useBox.getValue());
        }
    }
}
private void kill(FlowSet in, Unit u, FlowSet out) {
    FlowSet kills = (FlowSet)emptySet.clone();
    for (ValueBox defBox: u.getUseBoxes()) {
        if (defBox.getValue() instanceof Local) {
            for (BinopExpr e: in) {
                for (ValueBox useBox: e.getUseBoxes()) {
                    if (useBox.getValue() instanceof Local
                        && useBox.getValue().equivTo(defBox.getValue()))
                        kills.add(e);
                }
            }
        }
    }
    in.difference(kills, out);
}
protected Object entryInitialFlow() {
    return new ValueArraySparseSet();
}

protected Object newInitialFlow() {
    return new ValueArraySparseSet();
}
Example 3: Branched Nullness

A local variable $v$ is non-null at $s$ if all control-flow paths reaching $s$ result in $v$ being assigned a value different from null.

```
fx = new foo()
{
    y
}
{
    x, y
}
{
    x
}
{
    x, b
}
if (y == null)
{
    x.bar()
y = null
    y = b.f
    {x}
    {x, y}
    {x, b}
    {x}
```
HOWTO: Soot Flow Analysis

Again, here’s what to do:

1. Subclass `*FlowAnalysis`
2. Implement abstraction: `merge()`, `copy()`
3. Implement flow function `flowThrough()`
4. Implement initial values:
   - `newInitialFlow()` and `entryInitialFlow()`
5. Implement constructor
   - (it must call `doAnalysis()`)

Analyzing Java Programs with Soot – p. 65/89
Step 1: Forward or Backward?

Nullness is a branched forward flow analysis, since flow $f^n$ computes OUT sets from IN sets, sensitive to branches.

Now subclass `ForwardBranchedFlowAnalysis`.

```java
class NullnessAnalysis extends ForwardBranchedFlowAnalysis<Unit, FlowSet> {
```
Step 2: Abstraction domain

Domain: sets of Locals known to be non-null
Partial order is subset inclusion.

(More complicated abstractions possible* for this problem; e.g. ⊥, ⊤, null, non-null per-local.)

This time, use ArraySparseSet to implement:

```java
void merge(FlowSet in1, FlowSet in2, FlowSet out);
void copy(FlowSet src, FlowSet dest);
```

* see soot.jimple.toolkits.annotation.nullcheck.BranchedRefVarsAnalysis
Implementing an Abstraction

For a forward analysis, `copy` and `merge` mean:

- `copy(dir)` brings OUT set to predecessor’s IN set.
- `merge(dir)` joins two OUT sets to make an IN set.
Step 2: copy() for nullness

Same as for live locals.

protected void copy(FlowSet src,
                    FlowSet dest) {
    src.copy(dest);
}

Use copy() method from FlowSet.
Step 2: merge() for nullness

In branched nullness, a variable $v$ is non-null if it is non-null on all paths from start to $s$, so we use intersection.

Like copy(), use FlowSet method – here, intersection():

```java
void merge(...) {
    srcSet1.intersection(srcSet2, destSet);
}
```
Step 3: Branched Flow Function

Need to differentiate between branch and fall-through OUT sets.

```java
protected void flowThrough(     FlowSet srcValue,     Unit unit,     List<FlowSet> fallOut,     List<FlowSet> branchOuts)
```

`fallOut` is a one-element list.
`branchOuts` contains a FlowSet for each non-fallthrough successor.
Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.
Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.
- Remove kill set (defined *Locals*).

\[
y \text{ in } y = y \cdot \text{next};
\]
Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.
- Remove kill set (defined `Locals`).
  \[ y \text{ in } y = y.\text{next}; \]
- Add gen set.
  \[ x \text{ in } x.\text{foo}(); \]
Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.
- Remove kill set (defined locals).
  \[ y \text{ in } y = y\.next; \]
- Add gen set.
  \[ x \text{ in } x\.f00(); \]
- Handle copy statements.
Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.
- Remove kill set (defined `Locals`).
  ```java
  y in y = y.next;
  ```
- Add gen set.
  ```java
  x in x.foo();
  ```
- Handle copy statements.
- Copy to branch and fallthrough lists.
Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.
- Remove kill set (defined `Locals`).
  
  ```java
  y in y = y.next;
  ```
- Add gen set.
  
  ```java
  x in x.foo();
  ```
- Handle copy statements.
- Copy to branch and fallthrough lists.
- Patch sets for `if` statements.
Step 4: Initial values

Initialize IN, OUT sets.

- Create initial sets (\(\top\) from constr.)
  
  ```java
  FlowSet newInitialFlow() {
    return fullSet.clone();
  }
  ```

- Create entry sets (emptySet from constr.)
  
  ```java
  FlowSet entryInitialFlow() {
    return emptySet.clone();
  }
  ```

(To be created in constructor!)
Step 5: Constructor: Prologue

Create auxiliary objects.

```java
public NullnessAnalysis(UnitGraph g) {
    super(g);

    unitToGenerateSet = new HashMap();
    Body b = g.getBody();
}
```
Step 5: Constructor: Finding All Locals

Create flowsets, finding all locals in body:

```java
emptySet = new ArraySparseSet();
fullSet = new ArraySparseSet();

for (Local l : b.getLocals()) {
    if (l.getType().instanceof RefLikeType)
        fullSet.add(l);
}
```
Step 5: Creating gen sets

Precompute, for each statement, which locals become non-null after execution of that stmt.

- \( x \) gets non-null value:
  \( x = * \), where \( * \) is NewExpr, ThisRef, etc.

- successful use of \( x \):
  \( x.f, x.m() \), entermonitor \( x \), etc.
Don’t forget to call `doAnalysis()`!

```java
... doAnalysis();
...}
```
To instantiate a branched analysis & collect results:

```java
NullnessAnalysis na=new NullnessAnalysis(b);

// a SparseArraySet of non-null variables.
na.getFlowBefore(s);

// another SparseArraySet
if (s.fallsThrough()) na.getFallFlowAfter(s);

// a List of SparseArraySets
if (s.branches()) na.getBranchFlowAfter(s);
```
Adding transformations to Soot (easy way)

1. Implement a `BodyTransformer` or a `SceneTransformer`
   - `internalTransform` method does the transformation

2. Choose a pack for your transformation (usually `jtp`)

3. Write a `main` method that adds the transform to the pack, then runs Soot’s `main`
Want to run a set of Transformer objects with one method call.

⇒ Group them in a Pack.

Soot defines default Packs which are run automatically. To add a Transformer to the jtp Pack:

```java
Pack jtp = G.v().PackManager().getPack("jtp");
jtp.add(new Transform("jtp.nt",
    new NullTransformer()));
jtp.add(new Transform("jtp.nac",
    new NullnessAnalysisColorer()));
```
Running Soot more than once

- All Soot global variables are stored in `G.v()`
- `G.reset()` re-initializes all of Soot
Generating Jimple

```
.class

coffi

.jimple

Jimple parser

jb

Jimple
```

Analyzing Java Programs with Soot – p. 82/89
The p is sometimes silent.

Soot Pack Naming Scheme

\[ w? (j | s | b | g) (b | t | o | a) p \]

- **w** \( \rightarrow \) Whole-program phase
- **j, s, b, g** \( \rightarrow \) Jimple, Shimple, Baf, Grimp
- **b, t, o, a** \( \rightarrow \)
  - (b) Body creation
  - (t) User-defined transformations
  - (o) Optimizations with -O option
  - (a) Attribute generation
jb converts naive Jimple generated from bytecode into typed Jimple with split variables
Soot Packs (Jimple)

**jtp** performs user-defined intra-procedural transformations

**jop** performs intra-procedural optimizations
- CSE, PRE, constant propagation, . . .

**jap** generates annotations using whole-program analyses
- null-pointer check
- array bounds check
- side-effect analysis
Soot Packs (Back-end)

bb performs transformations to create Baf
bop performs user-defined Baf optimizations

gb performs transformations to create Grimp
gop performs user-defined Grimp optimizations

tag aggregates annotations into bytecode attributes
Conclusion

- Have introduced Soot, a framework for analyzing, optimizing, (tagging and visualizing) Java bytecode.
- Have shown the basics of using Soot as a stand-alone tool and also how to add new functionality to Soot.
- Now for some homework and reading.
Resources

Main Soot page: www.sable.mcgill.ca/soot/

Theses and papers:
  www.sable.mcgill.ca/publications/

Tutorials: www.sable.mcgill.ca/soot/tutorial/

Javadoc: in main Soot distribution,
  www.sable.mcgill.ca/software/#soot and also online at www.sable.mcgill.ca/soot/doc/.

Mailing lists:
  www.sable.mcgill.ca/soot/#mailingLists

Soot in a Course:
  www.sable.mcgill.ca/~hendren/621/