Soot, a Tool for Analyzing and Transforming Java Bytecode

Laurie Hendren, Patrick Lam, Jennifer Lhoták, Ondřej Lhoták and Feng Qian
McGill University

Special thanks to John Jorgensen and Navindra Umanee for help in preparing
Soot 2.0 and this tutorial.

Soot development has been supported, in part, by research grants from NSERC,
FCAR and IBM

http://www.sable.mcgill.ca/soot/
ACT I (*Warming Up*):
- Introduction and Soot Basics (*Laurie*)
- Intraprocedural Analysis in Soot (*Patrick*)

ACT II (*The Home Stretch*):
- Interprocedural Analyses and Call Graphs (*Ondřej*)
- Attributes in Soot and Eclipse (*Ondřej, Feng, Jennifer*)
- Conclusion, Further Reading & Homework (*Laurie*)
Introduction and Soot Basics

- What is Soot?
- Soot: Past and Present
- Soot Overview
- IRs: Baf, Jimple, Shimple, Grimp, Dava
- Soot as an end-user tool and Soot as an Eclipse plugin

... switching gears ....

- Jimple and Soot Implementation Basics
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
What is Soot? (2)

- 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.
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 have just been released - June 2003 - JIT for PLDI 2003.

This tutorial is based on Soot 2.0.
Soot Overview

**Java** source
- **javac**

**SML** source
- **MLJ**

**Scheme** source
- **KAWA**

**Eiffel** source
- **SmallEiffel**

Eclipse

**class files**

**SOOT**

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

**Optimized class files + attributes**

**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
Soot as an end-user tool: Command-line

1. Install Java.

2. Download two .jar files (one for soot and one for jasmin) and put them on your CLASSPATH.

   java soot.Main --help
   List options.

   java soot.Main --version
   Print version information.
Command-line: processing classes

\texttt{java soot.Main Foo}
\hspace{1em} Process \texttt{Foo.class} in the current directory and produce a new class file in \texttt{sootOutput/Foo.class}.

\texttt{java soot.Main \text{-f} jimple Foo}
\hspace{1em} Same as above, but produce Jimple in \texttt{sootOutput/Foo.jimple}.

\texttt{java soot.Main \text{-f} dava Foo}
\hspace{1em} Decompile \texttt{Foo.class} and produce \texttt{Foo.java} in \texttt{sootOutput/dava/src/Foo.java}.
java soot.Main -O Foo
   Run intraprocedural optimizations and produce optimized Foo.class.

java soot.Main -O --app Foo
   Run intraprocedural optimizations on Foo.class and all application classes reachable from Foo.class.

java soot.Main -W --app Foo
   Perform whole program analysis and produce optimized classes for Foo.class and all application classes reachable from Foo.
Command-line: a more complex example

java soot.Main -W -app -f jimple
  -p jb use-original-names:true
  -p cg.spark on
  -p cg.spark simplify-offline:true
  -p jop.cse on
  -p wjop.smb on -p wjop.si off
  Foo

Starting at Foo.class, process all reachable classes in an interprocedural fashion and produce Jimple as output for all application classes.
Command-line: a more complex example

```
java soot.Main -W -app -f jimple
-p jb use-original-names:true
-p cg.spark on
-p cg.spark simplify-offline:true
-p jop.cse on
-p wjop.smb on -p wjop.si off
Foo
```

When producing the original Jimple from the class files, keep the original variable names, if available in the attributes (i.e. class file produced with `javac -g`).
Command-line: a more complex example

```java
java soot.Main -W -app -f jimple
-p jb use-original-names:true
-p cg.spark on
-p cg.spark simplify-offline:true
-p jop.cse on
-p wjop.smb on -p wjop.si off
Foo
```

Use Spark for points-to analysis and call graph, with Spark simplifying the points-to problem by collapsing equivalent variables. Note: on is a short form for enabled:true.
Command-line: a more complex example

```
java soot.Main -W -app -f jimple
-p jb use-original-names:true
-p cg.spark on
-p cg.spark simplify-offline:true
-p jop.cse on
-p wjop.smb on -p wjop.si off
```

Foo

Turn on the intra and interprocedural optimizations phases (-W).
Enable *common sub-expression elimination* (cse).
Enable *static method binding* (smb) and disable *static inlining* (si).
Soot as an end-user tool: Eclipse Plugin

1. Install Java

2. Install Eclipse www.eclipse.org

3. Download one .jar file and unjar it into your Eclipse plugin directory

4. Start Eclipse

- IDE-based optimization, decompilation and visualization
- GUI for setting and storing Soot option configurations
- tooltips for documentation on options
- Eclipse views for Soot IRs
Switching Gears ... Let’s get dirty

Now we want to understand:

- details of Jimple
- internal workings of Soot

To work with Soot in this way, you should download the complete package `soot-2.0.jar` which contains the complete Java source, class files, Javadoc documentation, Soot tutorials, source and compiled forms of the plugin, and our modified jasmin assembler.
Jimple is:

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

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 → 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.

<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>
Bytecode → Jimple

- Performed in the \( j_b \) phase.
- Makes a 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 and packs locals
- provides a good starting point for analysis and optimization
Jimple → Bytecode

- Performed in the \texttt{bb} or \texttt{gb} phase.

- 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.

- Second approach works better and produces very good bytecode.

- Produces bytecode that is different than what \texttt{javac} produces, breaks immature JITs.
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.
Soot Classes

- Scene
  - Scene.v()
  - getSootClass()
- SootClass
  - getField()
  - getMethod()
- SootMethod
  - getSignature()
  - getActiveBody()
- SootField
  - getSignature()
- JimpleBody

(singleton)
Body-centric View

SootMethod

JimpleBody

getActiveBody()

getLocals()

getUnits()

Chain

getTraps()

Chain

Chain
Getting a UnitGraph

- SootMethod
  - getBody()
  - getActiveBody()
  - getLocals()
  - getTraps()
  - getUnits()

- JimpleBody
  - new BriefUnitGraph()
  - getUnits()

- UnitGraph
  - getUnits()

- Chain
  - getBody()
  - getLocals()
  - getTraps()
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
  
  (load x)

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

- **Stmt**: also used in Grimp
  
  (z = x + y * 2 % n;)

Soot, a Tool for Analyzing and Transforming Java Bytecode – p. 32/148
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?

\[ s: \quad x = \boxed{y \text{ op } z} \]

AssignStmt

\[ \text{VB} \quad \text{VB} \]

\[ \text{OpExpr} \]

\[ \text{VB} \quad \text{VB} \]

\[ \mathbf{x} \quad \mathbf{y} \quad \mathbf{z} \]
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: \( x = [y \text{ op } z] \);
```

`getDefBoxes(ut) = \{x\}`

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

- `getValue()`: Dereferencing a pointer.
  
  \[ x \rightarrow x \]

- `setValue()`: mutates the value in the `Box`. 
On UseBoxes

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.

```
ut: \textcolor{red}{x} = \boxed{y \text{ op } z};
```

getUseBoxes(ut) = \{\boxed{y}, \boxed{z}, \boxed{y \text{ op } z}\}

(List containing 3 ValueBoxes, 2 containing Locals & 1 Expr)
Change all instances of $y$ to 1:

```
AssignStmt
  VB
  VB
  x
  OpExpr
    VB
    VB
    y
    z
setValue()
```

```
AssignStmt
  x
  OpExpr
    y
    z
??
```
/* 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 \textit{Values}:

- \texttt{containsArrayRef()}, \texttt{getArrayRef()}, \texttt{getArrayRefBox()}
- \texttt{containsInvokeExpr()}, \texttt{getInvokeExpr()}, \texttt{getInvokeExprBox()}
- \texttt{containsFieldRef()}, \texttt{getFieldRef()}, \texttt{getFieldRefBox()}

\textbf{More Abstract Accessors: Stmt}
Program and Cast

ACT I (Warming Up):
- Introduction and Soot Basics (Laurie)
- Intraprocedural Analysis in Soot (Patrick)

ACT II (The Home Stretch):
- Interprocedural Analyses and Call Graphs (Ondřej)
- Attributes in Soot and Eclipse (Ondřej, Feng, Jennifer)
- Conclusion, Further Reading & Homework (Laurie)
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, a Tool for Analyzing and Transforming Java Bytecode – p. 45/148
Soot provides you with:

- **impls 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.
Flow Analysis Hierarchy

AbstractFlowAnalysis

FlowAnalysis

Forward- Backward-

BranchedFlowAnalysis

Forward-
Soot Flow Analyses

AbstractFlowAnalysis

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

BranchedFlowAnalysis
  - Forward-
    - Casts
    - Nullness
A forward analysis computes OUT from IN:

\[
\begin{align*}
\text{flow dir} & \quad i \\
\text{s} & \quad f_s(i) \\
\text{t} & \quad f_t(f_s(i))
\end{align*}
\]

A backward analysis computes IN from OUT:

\[
\begin{align*}
\text{flow dir} & \quad f_t(i) \\
\text{s} & \quad f_s(f_t(i)) \\
\text{t} & \quad f_t(i)
\end{align*}
\]
Outline: Soot Flow Analysis Examples

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

- live locals
- branched nullness testing
Running 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 $v$.
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()`)

Soot, a Tool for Analyzing and Transforming Java Bytecode – p. 52/148
Live variables is a backward flow analysis, since flow $f^n$ computes IN sets from OUT sets.

In Soot, we subclass `BackwardFlowAnalysis`.

```java
class LiveVariablesAnalysis extends BackwardFlowAnalysis
```

soot.toolkits.scalar.BackwardFlowAnalysis
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 \texttt{ArraySparseSet}
implementation of \texttt{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.
More on Implementing an Abstraction

Signatures:

```java
void merge(Object src1, Object src2, Object dest);
void copy(Object src, Object dest);
```

We delegate implementation to FlowSet.
Flow Sets and Soot

Using a FlowSet is not mandatory, but 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);```

Soot, a Tool for Analyzing and Transforming Java Bytecode – p. 57/148
Digression: types of FlowSets

Which FlowSet do you want?

- **ArraySparseSet**: simple list
  
  | foo | bar | z |
  |
  (simplest possible)

- **ArrayPackedSet**: bitvector w/ map
  
  | 00100101 | 10101111 | 10000000 |

  (can complement, need universe)

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

protected void copy(Object src, Object dest) {
    FlowSet sourceSet = (FlowSet) src,
    destSet = (FlowSet) dest;

    sourceSet.copy(destSet);
}

Use \texttt{copy()} method from \texttt{FlowSet}.
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`.

Like `copy()`, use FlowSet's `union`:

```java
void merge(...) {
    // [cast Objects to FlowSets]
    src1Set.union(src2Set, destSet);
}
```

One might also use `intersection()`, or implement a more exotic merge.
Step 3: Flow equations

Goal: At a unit like \( x = y \times z \):

\begin{verbatim}
  kill    def x;
  gen     uses y, z.
\end{verbatim}

How? Implement this method:

```java
protected void flowThrough
    (Object srcValue,
     Object u,
     Object destValue)
```
Step 3: Casting

Soot’s flow analysis framework is polymorphic. Need to cast to do useful work.

Start by:

- **casting** srcValue, destValue to FlowSet.
- **casting** u to Unit ut.

In code:

```java
FlowSet src = (FlowSet)srcValue,
    dest = (FlowSet)destValue;
Unit ut = (Unit)u;
```
Step 3: Copying

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

```java
src.copy (dest);
```

Use \texttt{FlowSet} methods.
Step 3: Implementing flowThrough

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

\[
\text{IN[ut]} = \text{flowThrough}(\text{OUT[ut]})
\]
\[
= \text{OUT[ut]} \setminus \text{kills[ut]} \cup \text{gens[ut]}
\]

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

// 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
  Object newInitialFlow()
  {
    return new ArraySparseSet();
  }
  ```

  **Create initial sets for exit nodes**
  
  ```java
  Object entryInitialFlow()
  {
    return new ArraySparseSet();
  }
  ```

  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 SparseArraySets
// of live variables:
    lv.getFlowBefore(s);
    lv.getFlowAfter(s);
```
Running Example 2: 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 \( \text{null} \).
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()`)

Soot, a Tool for Analyzing and Transforming Java Bytecode – p. 72/148
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 {
```

Soot, a Tool for Analyzing and Transforming Java Bytecode – p. 73/148
Step 2: Abstraction domain

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

(More complicated abstractions possible* for this problem; e.g. \bot, \top, \texttt{null}, \texttt{non-null} per-local.)

Again use \texttt{ArraySparseSet} to implement:

```java
void merge(Object in1, Object in2,
           Object out);

void copy(Object src, Object dest);
```

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

For a forward analysis, \texttt{copy} and \texttt{merge} mean:

\begin{align*}
\text{\texttt{copy} (\text{dir})} & \quad \text{dest} \leftarrow \text{src} \quad \text{CFG} \\
\text{\texttt{merge} (\text{dir})} & \quad \text{dest} = \text{src}_1 \Join \text{src}_2
\end{align*}

\texttt{copy()} brings OUT set to predecessor’s IN set.
\texttt{merge()} joins two OUT sets to make an IN set.
Same as for live locals.

```java
protected void copy(Object src, Object dest) {
    FlowSet sourceSet = (FlowSet) src,
    destSet = (FlowSet) dest;

    sourceSet.copy(destSet);
}
```

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(...) {
    // [cast Objects to FlowSets]
    srcSet1.intersection(srcSet2, destSet);
}
```
Step 3: Branched Flow Function

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

```java
protected void flowThrough(Object srcValue, Unit unit, List fallOut, List 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.
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;
```
We do the following things in our flow function:

- Create copy of src set.
- Remove kill set (defined \texttt{Locals}).
  
  \begin{verbatim}
  y in y = y.next;
  \end{verbatim}

- Add gen set.
  
  \begin{verbatim}
  x in x.foo();
  \end{verbatim}
Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.
- Remove kill set (defined $\text{Locals}$).
  ```
  y in y = y.next;
  ```
- Add gen set.
  ```
  x in x.foo();
  ```
- 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`).
  ```
  y in y = y.next;
  ```
- Add gen set.
  ```
  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`).
  \[
  y \text{ in } y = y.\text{next};
  \]
- Add gen set.
  \[
  x \text{ in } x.\text{foo}();
  \]
- 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
  Object newInitialFlow() {
    return fullSet.clone();
  }
  ```

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

(To be created in constructor!)
Create auxiliary objects.

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

    unitToGenerateSet = new HashMap();
    Body b = g.getBody();
```
Create flowsets, finding all locals in body:
emptySet = new ArraySparseSet();
fullSet = new ArraySparseSet();

for (Local l : b.getLocals()) {
    if (l.getType().
        instanceof RefLikeType)
        fullSet.add(l);
}

Step 5: Constructor: Finding All Locals
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 \texttt{NewExpr}, \texttt{ThisRef}, etc.

- successful use of \( x \):
  - \( x.f, x.m() \), \texttt{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

4. (Optional) If your transformation needs command-line options, call `setDeclaredOptions()`
On Packs

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()));
```
Extending Soot (hard way)

Some don’t like calling `soot.Main.main()`. What does `main()` do?

1. `processCmdLine()`
2. `Scene.v().loadNecessaryClasses()`
3. `PackManager.v().runPacks()`
4. `PackManager.v().writeOutput()`

You can do any or all of these yourself:

- `Options.v()` contains setter methods for all options
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
- Jimple parser
- jb
- Jimple
Intra-procedural packs

Jimple

stp → sop → sap

jtp → jop → jap

bb → bop → tag

gb → gop

Dava

Jasmin

Shimple

Jimple

Baf

Grimp

Output
The p is sometimes silent.

Soot Pack Naming Scheme

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

- **w** ⇒ Whole-program phase
- **j, s, b, g** ⇒ Jimple, Shimple, Baf, Grimp
- **b, t, o, a** ⇒
  - (b) Body creation
  - (t) User-defined transformations
  - (o) Optimizations with -O option
  - (a) Attribute generation
Soot Packs (Jimple Body)

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
Program and Cast

ACT I (Warming Up):
- Introduction and Soot Basics (Laurie)
- Intraprocedural Analysis in Soot (Patrick)

ACT II (The Home Stretch):
- Interprocedural Analyses and Call Graphs (Ondřej)
- Attributes in Soot and Eclipse (Ondřej, Feng, Jennifer)
- Conclusion, Further Reading & Homework (Laurie)
Interprocedural Outline

- Soot’s whole-program mode
- Call graph
- Points-to information (Spark)
  - (Spark was my M.Sc. thesis)
Soot’s whole-program mode

- Use \(-w\) switch for whole-program mode
- Enables cg, wjtp, wjap packs
- Whole-program information from these packs available to rest of Soot through Scene
  - Call graph
  - Points-to information
- Whole program analyzed; only application classes written out, not library classes
- To also enable wjop, use \(-W\)
  - Method inlining, static binding
Soot Packs (Whole Program)

- **cg**: generates a call graph using CHA or more precise methods
- **wjtp**: performs user-defined whole-program transformations
- **wjop**: performs whole-program optimizations
  - static inlining
  - static method binding
- **wjap**: generates annotations using whole-program analyses
  - rectangular array analysis
Soot phases
Call Graph

- Collection of edges representing **all** method invocations known to Soot
  - explicit method invocations
  - implicit invocations of static initializers
  - implicit calls of `Thread.run()`
  - implicit calls of finalizers
  - implicit calls by `AccessController`
  - ...

- **Filter** can be used to select specific kinds of edges
Call Graph Edge

- Each Edge contains
  - Source method
  - Source statement (if applicable)
  - Target method
  - Kind of edge

<table>
<thead>
<tr>
<th>source m.</th>
<th>source stmt.</th>
<th>target m.</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>VIRTUAL</td>
</tr>
</tbody>
</table>

```java
foo() {
  o.bar();
  bar() {
    /* */
  }
}
```
/** Due to explicit invokevirtual instruction. */
public static final int VIRTUAL = 2;
/** Due to explicit invokeinterface instruction. */
public static final int INTERFACE = 3;
/** Due to explicit invokespecial instruction. */
public static final int SPECIAL = 4;
/** Implicit call to static initializer. */
public static final int CLINIT = 5;
/** Implicit call to Thread.run() due to Thread.start() call. */
public static final int THREAD = 6;
/** Implicit call to Thread.exit(). */
public static final int EXIT = 7;
/** Implicit call to non-trivial finalizer from constructor. */
public static final int FINALIZE = 8;
/** Implicit call to run() through AccessController.doPrivileged(). */
public static final int PRIVILEGED = 9;
/** Implicit call to constructor from java.lang.Class.newInstance(). */
public static final int NEWINSTANCE = 10;
Querying Call Graph

**edgesOutOf(SootMethod)** iterates over edges with a given source method.

**edgesOutOf(Unit)** iterates over edges with a given source statement.

**edgesInto(SootMethod)** iterates over edges with a given target method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
<th>Source Method</th>
<th>Target Method</th>
<th>Call Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>o.foo();</td>
<td></td>
<td>C1.foo()</td>
<td>VIRTUAL</td>
</tr>
<tr>
<td>main()</td>
<td>o.goo();</td>
<td></td>
<td>C1.goo()</td>
<td>VIRTUAL</td>
</tr>
<tr>
<td>main()</td>
<td>o.goo();</td>
<td></td>
<td>C2.goo()</td>
<td>VIRTUAL</td>
</tr>
<tr>
<td>bar()</td>
<td>o.foo();</td>
<td></td>
<td>C2.foo()</td>
<td>VIRTUAL</td>
</tr>
</tbody>
</table>
**Querying Call Graph**

- `edgesOutOf(SootMethod)` iterator over edges with given source method
- `edgesOutOf(Unit)` iterator over edges with given source statement
- `edgesInto(SootMethod)` iterator over edges with given target method

<table>
<thead>
<tr>
<th>Method</th>
<th>Target</th>
<th>Class</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>main()</code></td>
<td><code>o.foo();</code></td>
<td><code>C1.foo()</code></td>
<td>VIRTUAL</td>
</tr>
<tr>
<td><code>main()</code></td>
<td><code>o.goo();</code></td>
<td><code>C1.goo()</code></td>
<td>VIRTUAL</td>
</tr>
<tr>
<td><code>main()</code></td>
<td><code>o.goo();</code></td>
<td><code>C2.goo()</code></td>
<td>VIRTUAL</td>
</tr>
<tr>
<td><code>bar()</code></td>
<td><code>o.foo();</code></td>
<td><code>C1.foo()</code></td>
<td>VIRTUAL</td>
</tr>
</tbody>
</table>
Adapters

Adapters make an iterator over edges into an iterator over

**Sources** source methods

**Units** source statements

**Targets** target methods

<table>
<thead>
<tr>
<th>src</th>
<th>stmt</th>
<th>tgt</th>
<th>kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>src₁</td>
<td>stmt₁</td>
<td>tgt₁</td>
<td>kind₁</td>
</tr>
<tr>
<td>src₂</td>
<td>stmt₂</td>
<td>tgt₂</td>
<td>kind₂</td>
</tr>
<tr>
<td>src₃</td>
<td>stmt₃</td>
<td>tgt₃</td>
<td>kind₃</td>
</tr>
</tbody>
</table>
void mayCall( SootMethod src ) {
    CallGraph cg =
        Scene.v().getCallGraph();
    Iterator targets =
        new Targets(cg.edgesOutOf(src));

    while( targets.hasNext() ) {
        SootMethod tgt =
            (SootMethod) targets.next();
        System.out.println( "" +
            src+" may call "+tgt );
    }
}
Reachable Methods

- **ReachableMethods** object keeps track of which methods are reachable from entry points

  - `contains(SootMethod)` tests whether method is reachable
  - `listener()` returns an iterator over reachable methods
ReachableMethods rm = Scene.v().getReachableMethods();

if( rm.contains( myMethod ) )
    // myMethod is reachable

Iterator it = rm.listener();
while( it.hasNext() ) {
    SootMethod method =
        (SootMethod) it.next();
    // method is reachable
}
Transitive Targets

- TransitiveTargets class takes a CallGraph and optional Filter to select edges

  iterator(SootMethod)  iterator over methods transitively called from given method

  iterator(Unit)  iterator over methods transitively called from targets of given statement
Points-to analysis

- Default points-to analysis assumes that any pointer can point to any object
- Spark provides variations of context-insensitive subset-based points-to analysis
  - Work in progress on context-sensitive analyses
Spark settings

- `p cg.spark on` turns on Spark
- Spark used for both call graph, and points-to information
- Default setting is on-the-fly call graph, field-sensitive, most efficient algorithm and data structures
- `p cg.spark vta` Spark as VTA
- `p cg.spark rta` Spark as RTA
PointsToAnalysis interface

reachingObjects(Local) returns PointsToSet of objects pointed to by a local variable

x = y

reachingObjects(SootField) returns PointsToSet of objects pointed to by a static field

x = C.f

reachingObjects(Local, SootField) returns PointsToSet of objects pointed to by given instance field of the objects pointed to by local variable

x = y.f
PointsToSet interface

possibleTypes() returns a set of the possible types of the objects in the points-to set

hasNonEmptyIntersection(PointsToSet) tells us whether two points-to sets may overlap (whether the pointers may be aliased)
If I want to know...

... the types of the receiver o in the call:

```java
Local o;
PointsToAnalysis pa = Scene.v().getPointsToAnalysis();
PointsToSet ptset = pa.reachingObjects(o);
java.util.Set types = ptset.possibleTypes();
```
If I want to know...

... whether \( x \) and \( y \) may be aliases in
\[
x.f = 5;
y.f = 6;
z = x.f;
\]

Local \( x, y; \)
PointsToSet \( xset = \)
\[
\text{pa.reachingObjects}( x );
\]
PointsToSet \( yset = \)
\[
\text{pa.reachingObjects}( y );
\]
if(\( xset \).hasNonEmptyIntersection(\( yset \))
// they’re possibly aliased
SideEffectTester interface

Reports side-effects of any statement, including calls

**newMethod(SootMethod)** tells the side-effect tester that we are starting a new method

**unitCanReadFrom(Unit,Value)** returns true if the Unit (statement) might read the Value

**unitCanWriteTo(Unit,Value)** returns true if the Unit (statement) might write the Value
## Implementations of SideEffectTester

### NaiveSideEffectTester
- is conservative
- does not use call graph or points-to information
- does not require whole-program mode

### PASideEffectTester
- uses current call graph
- uses current points-to information
  - this may be naive points-to information
Program and Cast

ACT I (Warming Up):
- Introduction and Soot Basics (Laurie)
- Intraprocedural Analysis in Soot (Patrick)

ACT II (The Home Stretch):
- Interprocedural Analyses and Call Graphs (Ondřej)
- Attributes in Soot and Eclipse (Ondřej, Feng, Jennifer)
- Conclusion, Further Reading & Homework (Laurie)
Motivation of Soot Attributes

- We often want to attach annotations to code
  - to convey low-level analysis results, such as register allocation or array bounds check elimination to a VM
  - to convey analysis results to humans
  - to record profiling information
- Soot provides a framework to support the embedding of custom, user-defined attributes in class files
Java class file attributes

- Attributes of `class_info`, `method_info`, `field_info`, and `Code_attribute` structures
- In fact: Code is an attribute of a method
- Standard attributes: `SourceFile`, `ConstantValue`, `Exceptions`, `LineNumberTable`, `LocalVariableTable`
- VM is required to ignore attributes it does not recognize
The VM spec defines the format of attributes:

```java
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

- `attribute_name_index`, the index of the attribute’s name in the class files’ *Constant Pool*
- `attribute_length`, the length of the attribute’s data
- `info`, an array of raw attribute data
Attributes and Soot (overview)

- Soot parses several standard attributes
- New attributes can be created and attached
- Users can design their own attribute format
Tags in Soot Internals

Soot

SootClass

SootField

SootMethod

Body

Unit

Java class file

class_info

field_info

method_info

Code_attribute

ValueBox
Hosts

Hosts are objects that can hold Tags:

```java
package soot.tagkit;
public interface Host {
    public void addTag (Tag t);
    public Tag getTag (String aName);
    public List getTags ();
    public void removeTag (String name);
    public boolean hasTag (String aName);
}
```

Implementations:
SootClass, SootField, SootMethod, Body, Unit, ValueBox
Tags are objects that can be attached to Hosts:

```java
package soot.tagkit;
public interface Tag {
    public String getName();
    public byte[] getValue() throws AttributeValueException;
    public String toString();
}
```

- **Attribute** attached to class file structures (class, field, method)
- **Generic tags** attached to *Units* or *ValueBoxes*
interface Tag
   byte[] getValue()

interface Attribute
   void setValue(byte[])

class GenericAttribute

abstract class JasminAttribute
   byte[] decode(......)
   String getJasminValue(......)

class CodeAttribute

class LineNumberTag
class BytecodeOffsetTag
class ArrayNullCheckTag
class FieldRWTTag
e tc ......
Special case: attributes of Code_attribute

- **TagAggregator** aggregates tags of Units/ValueBoxes to **CodeAttribute**
- **CodeAttribute** is a table of (pc, value) pairs in class file

```
......
a=b[i];
......
getfield
```

```
......
a=b[i];
......
getfield
```

```
Jimple

Baf

code_attribute

Jasmin

label1: iaload
label2: getfield
.code_attribute
NullCheckAttribute
%label1%BA==

TagAggregator

CodeAttribute.getJasminValue()
Choosing an Aggregator

One Jimple statement may translate to multiple bytecode instructions

Jimple

\[ x = y.f \]

Bytecode

\[
\begin{align*}
\text{load } y \\
\text{getfield } f \\
\text{store } x
\end{align*}
\]

Which instruction(s) should get the tags?
Choosing an Aggregator

**ImportantTagAggregator**
attaches tag to the “most important” instruction (field reference, array reference, method invocation)
- Used for array bounds check, null pointer check, side-effect attributes

**FirstTagAggregator**
attaches tag to the first instruction
- Used for line number table attribute

Easy to make your own . . .
public abstract class TagAggregator extends BodyTransformer {

      ......
    abstract boolean wantTag(Tag t);
    abstract void considerTag(Tag t, Unit u);
    abstract String aggregatedName();
    void internalTransform(Body b, ... ) {
              ......
    }
  }
}
abstract class ImportantTagAggregator extends TagAggregator {
    /** Decide whether this tag should be aggregated by this aggregator. */
    public abstract boolean wantTag(Tag t);

    /** Return name of the resulting aggregated tag. */
    public abstract String aggregatedName();
}
Howto for creating new attributes

- Create a new Tag class, decide which structure is the host
- If the tag is for Units, write a tag aggregator by extending `TagAggregator` or one of its subclasses
- Parse attributes in bytecode consumer
Example: nullness attribute

Step 1: create NullCheckTag

class NullCheckTag {
    public String getName() { return "NullCheckTag"; }
    private byte value = 0;
    public byte[] getValue() {
        byte[] bv = new byte[1];
        bv[0] = value;
        return bv;
    }
    public void toString() {
        return ((value==0)?"[not null]":"[unknown]");
    }
}
Step 2: attach tags to units after analysis

```java
boolean needCheck;
s.addTag(new NullCheckTag(needCheck));
```
Example: nullness attribute

Step 3: create a NullTagAggregator

```java
p.add(new Transform("tag.null",
    NullTagAggregator.v()));

class NullTagAggregator
    extends ImportantTagAggregator {

    public boolean wantTag(Tag t) {
        return (t instanceof NullCheckTag);
    }

    public String aggregatedName() {
        return "NullCheckAttribute";
    }

    }
```
Attributes of Code_attribute extends JasminAttribute which generates textual representation of (label, value) pairs:

```java
String getJasminValue(Map instToLabel);
```
e.g. "NullCheckAttribute":

```java
null_check_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    { u2 pc;
        u1 data;
    } [attribute_length/3];
}
```
Motivation of Soot Attributes in Eclipse

- The Soot - Eclipse plug-in provides a mechanism for viewing attribute information in visual ways within Eclipse.

- This can aid:
  - software visualization
  - program understanding
  - analysis debugging
Visual Representations

- Three visual representations of attribute information:
  - Text displayed in tooltips
  - Color highlighting of chunks of code
  - Pop-up links
String Tags

- **StringTags** attach a string of information to a Host.

```java
s.addTag(new StringTag(val+" : NonNull"));
```

- The Soot - Eclipse plug-in displays the string as a tooltip when the mouse hovers over a line of text in the Java editor and Jimple editor.
Color Tags

- **ColorTags** attach a color to a Host.

  ```java
  v.addTag(new ColorTag(ColorTag.GREEN));
  v.addTag(new ColorTag(255, 0, 0));
  ```

- The Soot - Eclipse plug-in highlights the background color of the text in the editor at the appropriate positions with the given color in the Jimple editor.
Link Tags

- **LinkTags** attach a string of information, and a link to another part of code to a **Host**.

```java
String text = "Target:"+m.toString();
Host h = m;
String cName = m.getDeclaringClass().getName();
s.addTag(new LinkTag(text, h, cName));
```

- The Soot - Eclipse plug-in displays link which jumps to another part of the code when clicked in the Jimple Editor.
Program and Cast

ACT I (Warming Up):
- Introduction and Soot Basics (Laurie)
- Intraprocedural Analysis in Soot (Patrick)

ACT II (The Home Stretch):
- Interprocedural Analyses and Call Graphs (Ondřej)
- Attributes in Soot and Eclipse (Ondřej, Feng, Jennifer)
- Conclusion, Further Reading & Homework (Laurie)
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.
Try out Soot

**Super easy:** Soot as a stand-alone tool, Eclipse plugin

**Easy:** implement a new intraprocedural analysis and generate tags for it.

**More challenging:** implement whole program analysis, toolkit or a new IR.

Please stay in touch, tell us how you are using Soot and contribute back any new additions you make.
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/
Further reading


Initial design of attributes: CC 2001

Array bounds checking elimination: Feng’s thesis, CC 2002
