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/`

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)

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

www.sable.mcgill.ca/soot/

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

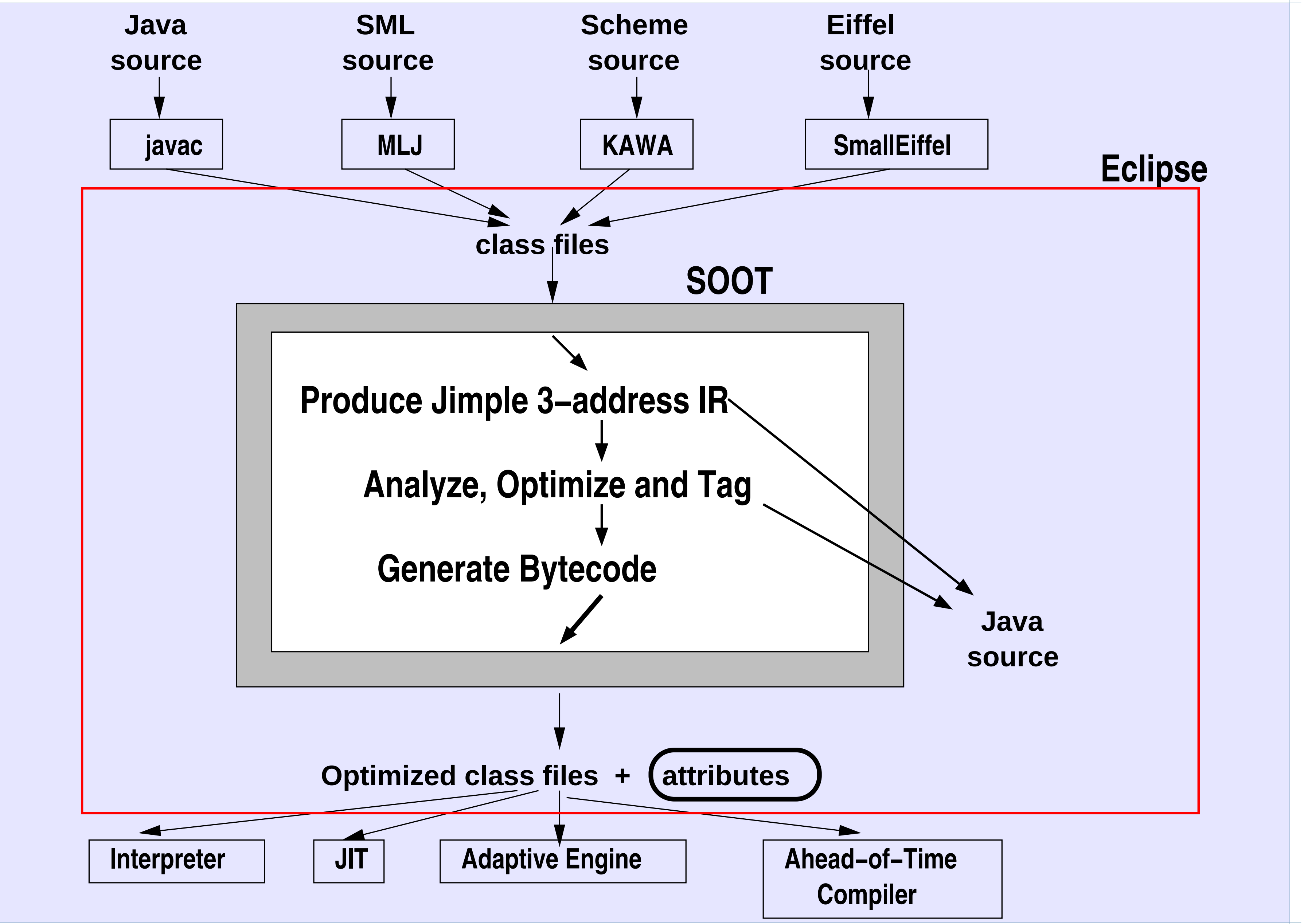
- Started in 1996-97 with the development of `coffi` by Clark Verbrugge and some first prototypes of `Jimple` IR by Clark and Raja Vallée-Rai.
- First publicly-available versions of Soot 1.x were associated with Raja's M.Sc. thesis
- New contributions and releases have been added by many graduate students at McGill and research results have been the topics of papers and theses.

www.sable.mcgill.ca/publications/

Soot: Past and Present (2)

- 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



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.

www.sable.mcgill.ca/software/#soot

Command-line: processing classes

`java soot.Main Foo`

Process `Foo.class` in the current directory and produce a new class file in `sootOutput/Foo.class`.

`java soot.Main -f jimple Foo`

Same as above, but produce Jimple in `sootOutput/Foo.jimple`.

`java soot.Main -f dava Foo`

Decompile `Foo.class` and produce `Foo.java` in `sootOutput/dava/src/Foo.java`.

Command-line: optimizing classes

`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 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

ww.sable.mcgill.ca/software/#soot

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

Jimple is:

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

aja's Thesis, CASCON99, CC2000, SAS2000

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 Jimple Stmts 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;
```

Context: other Jimple `Stmts`

```
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.

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 $\rightarrow$ Jimple

- Performed in the `jb` 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

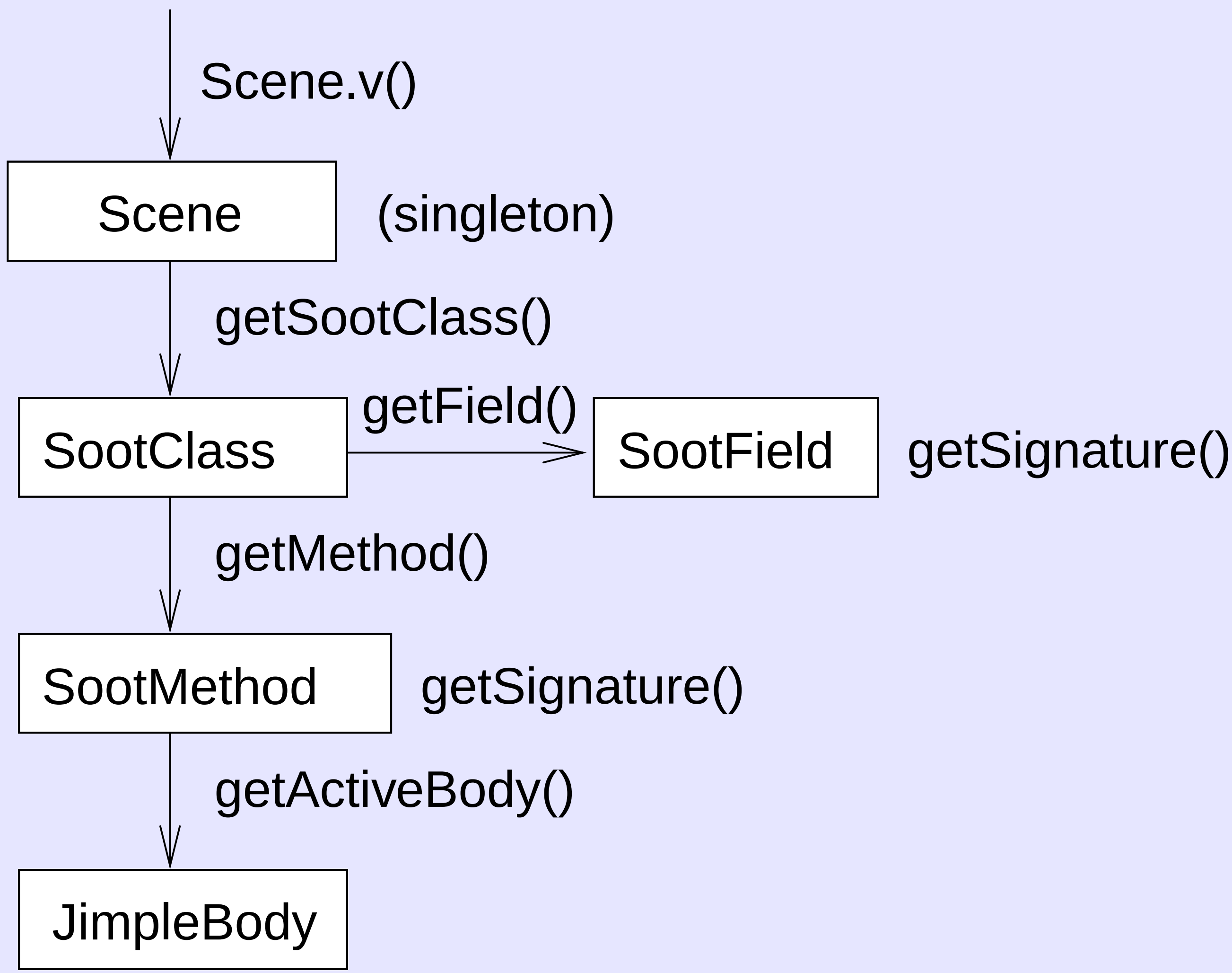
Jimple → Bytecode

- Performed in the bb or 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 `javac` produces, breaks immature JITs.

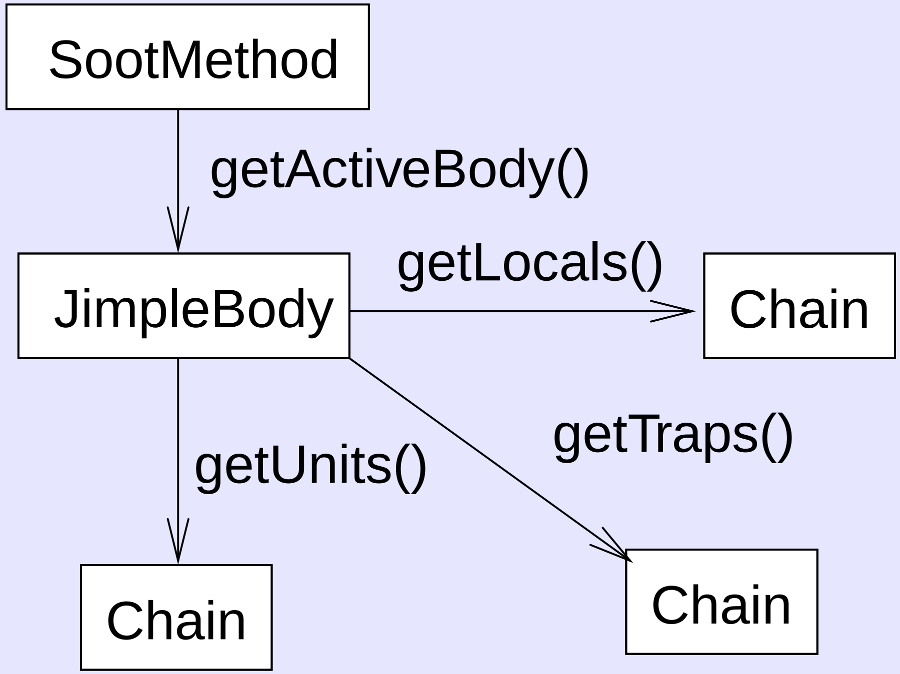
Soot Data Structure Basics

- 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.

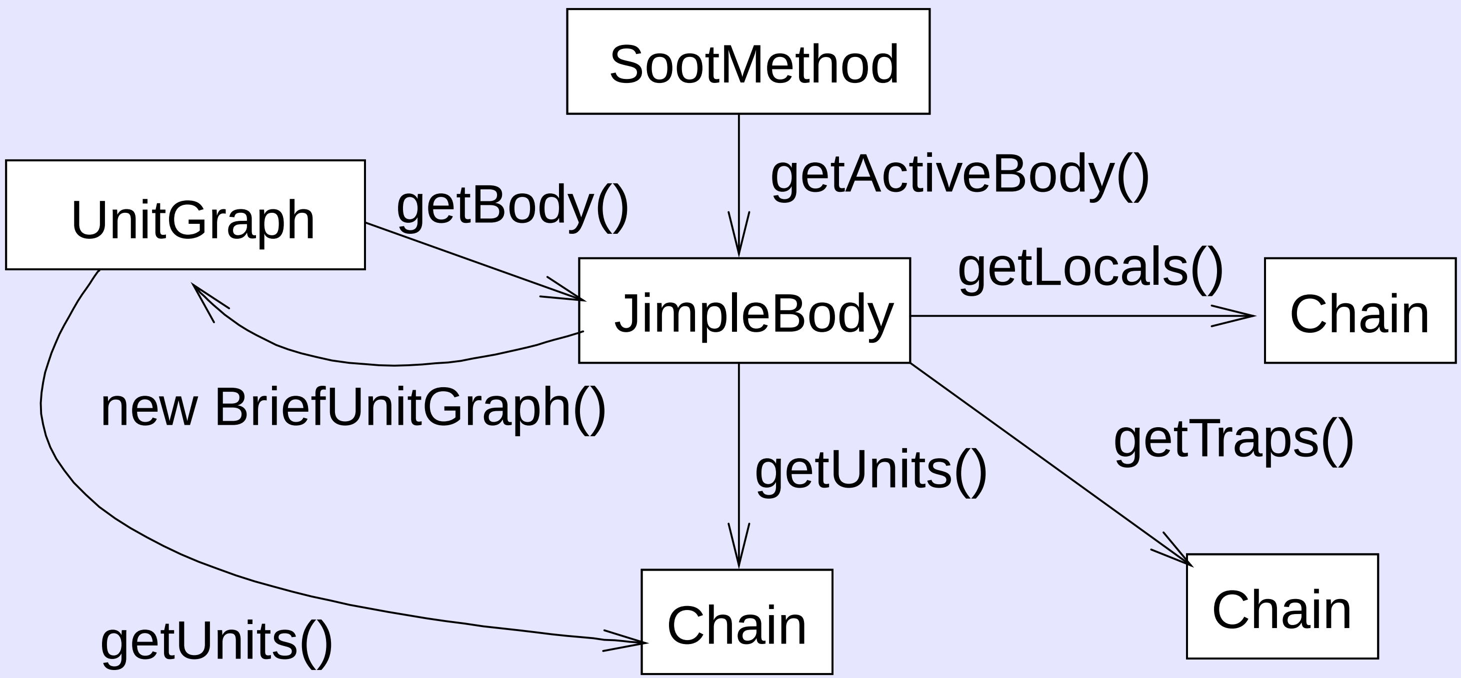
Soot Classes



Body-centric View



Getting a `UnitGraph`



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 Philosophy on `Units`

Accesses should be **abstract** whenever possible!

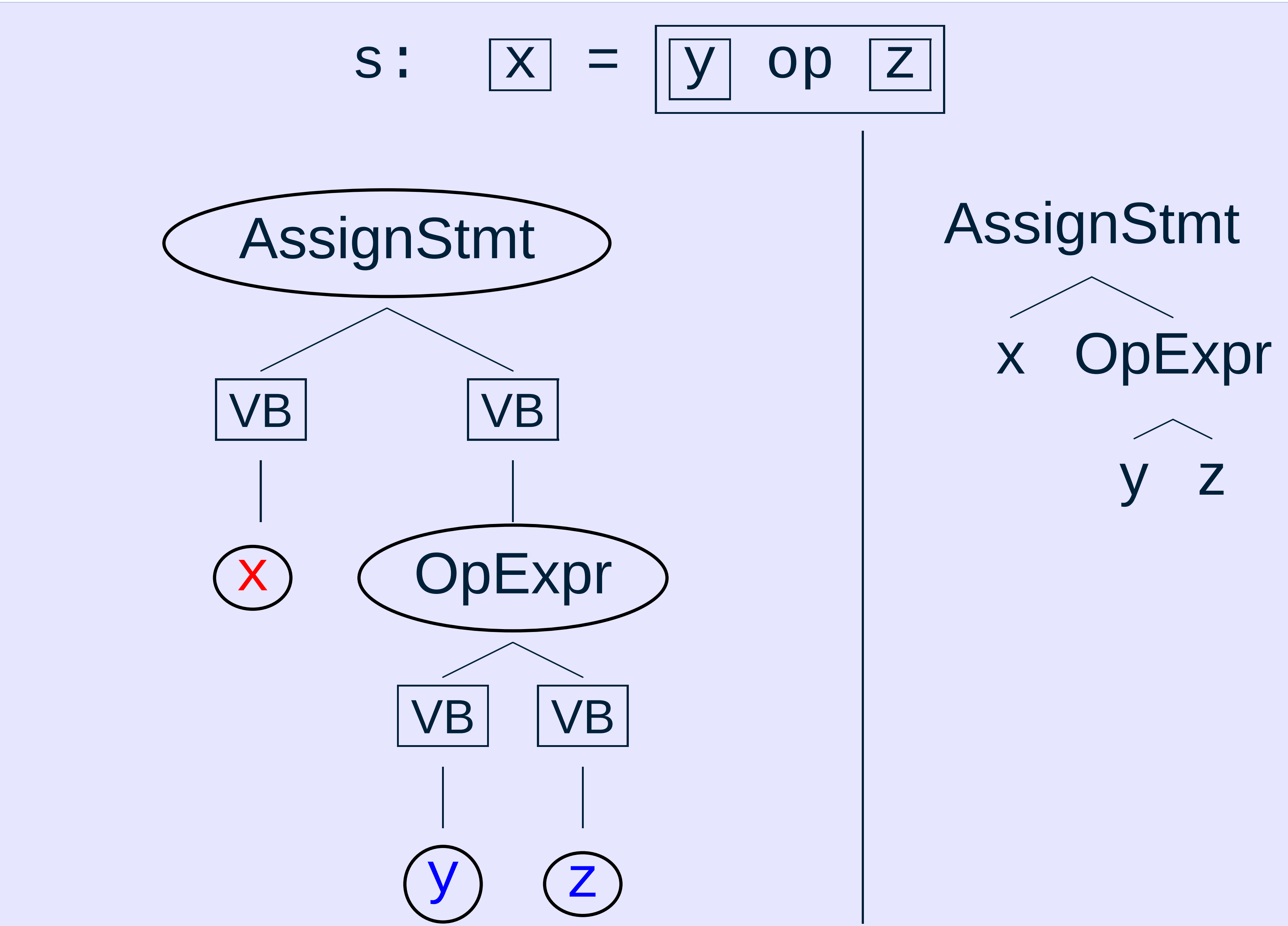
Accessing data:

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

(also control-flow information:)

```
fallsThrough(), branches(),
getBoxesPointingToThis(),
addBoxesPointingToThis(),
removeBoxesPointingToThis(),
redirectJumpsToThisTo()
```

What is a Box?


s: x = y op z
AssignStmt
VB
VB
x
OpExpr
VB
VB
y
z
AssignStmt
x OpExpr
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`)

On `Values` and `Boxes`

```
Value value = defBox.getValue();
```

- `getValue()`: Dereferencing a pointer.

  $\boxed{\mathtt{x}} \rightarrow \mathtt{x}$

- `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`.

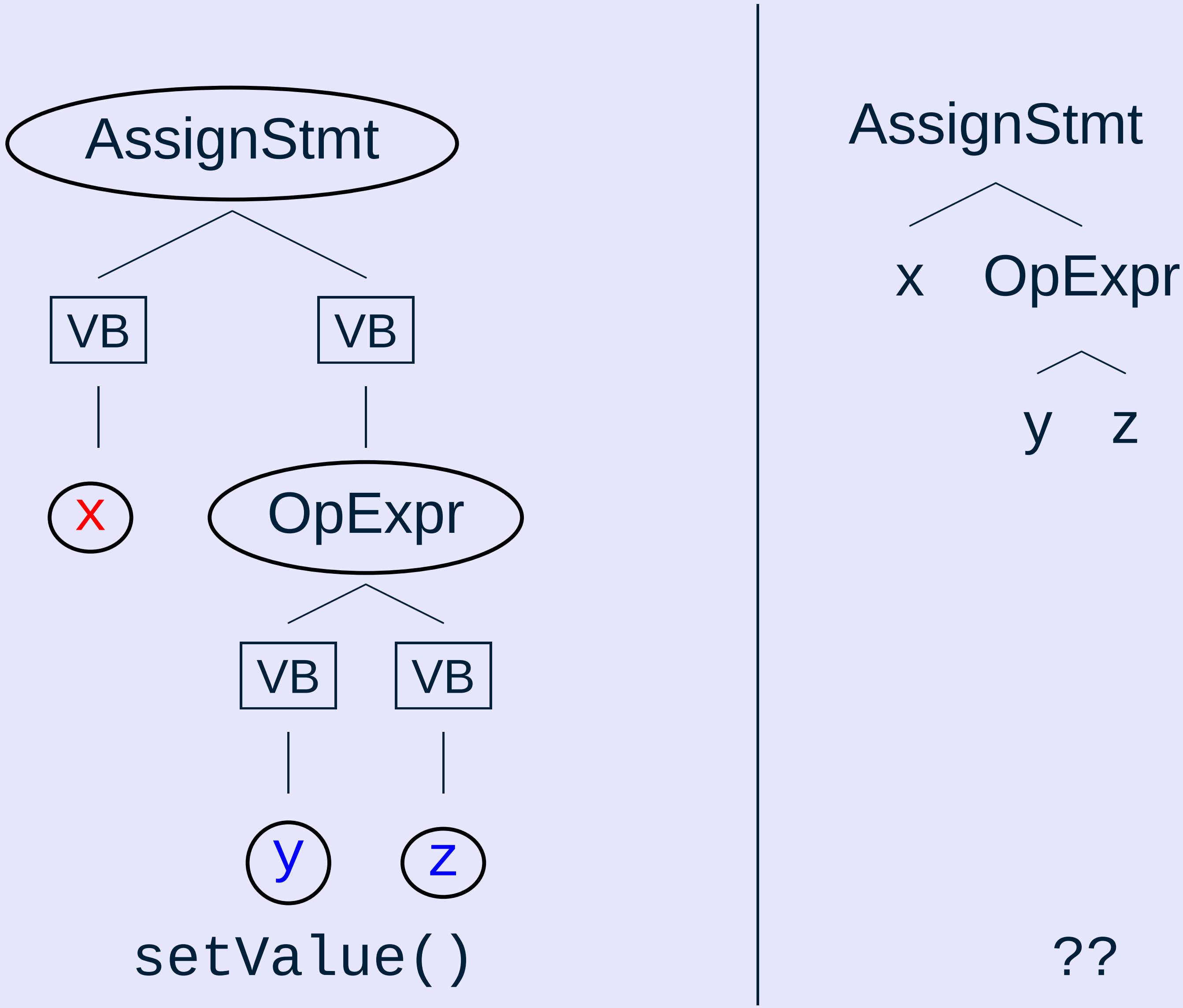
```
ut:   [x] = [[y] op [z]];

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`:



Search & Replace

```
/* 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));
```

More Abstract Accessors: `Stmt`

Jimple provides the following additional accessors for special kinds of `Values`:

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

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:

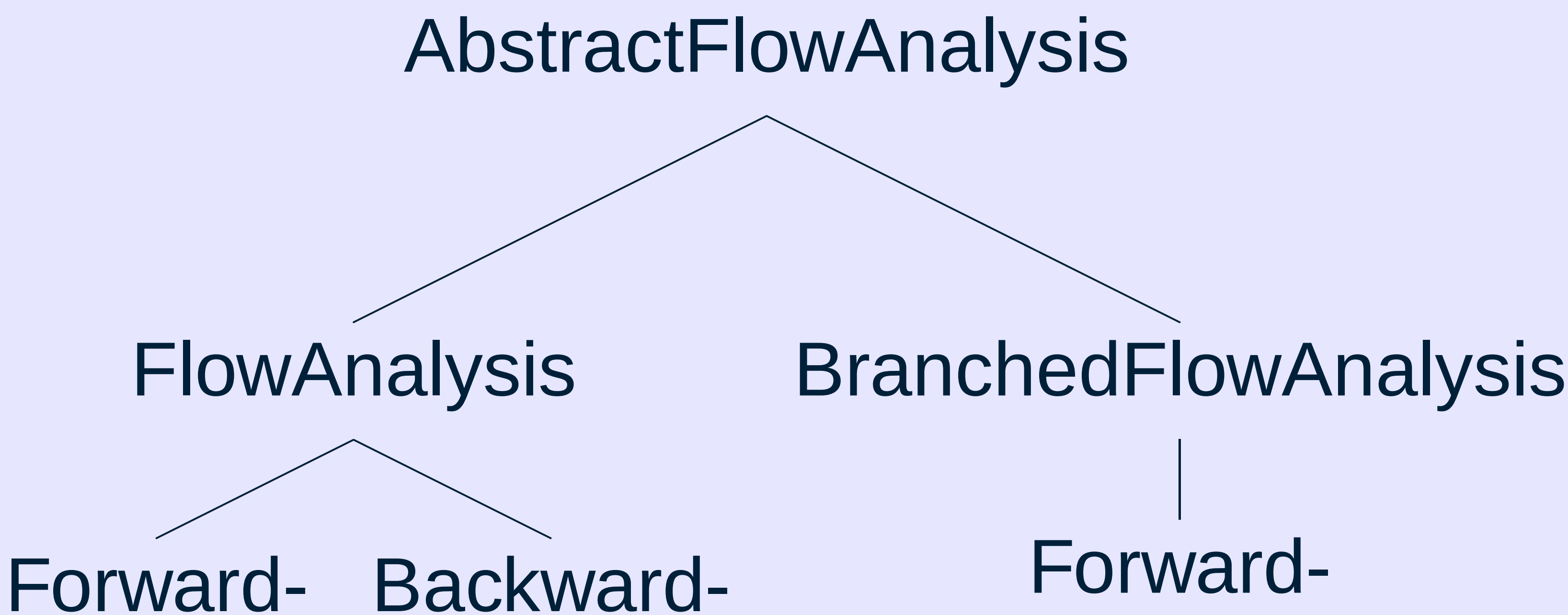
1. Subclass `*FlowAnalysis`
2. Implement abstraction: `merge()`, `copy()`
3. Implement flow function `flowThrough()`
4. Implement initial values: `newInitialFlow()` and `entryInitialFlow()`
5. Implement constructor (it must call `doAnalysis()`)

HOWTO: Soot Flow Analysis II

Soot provides you with:

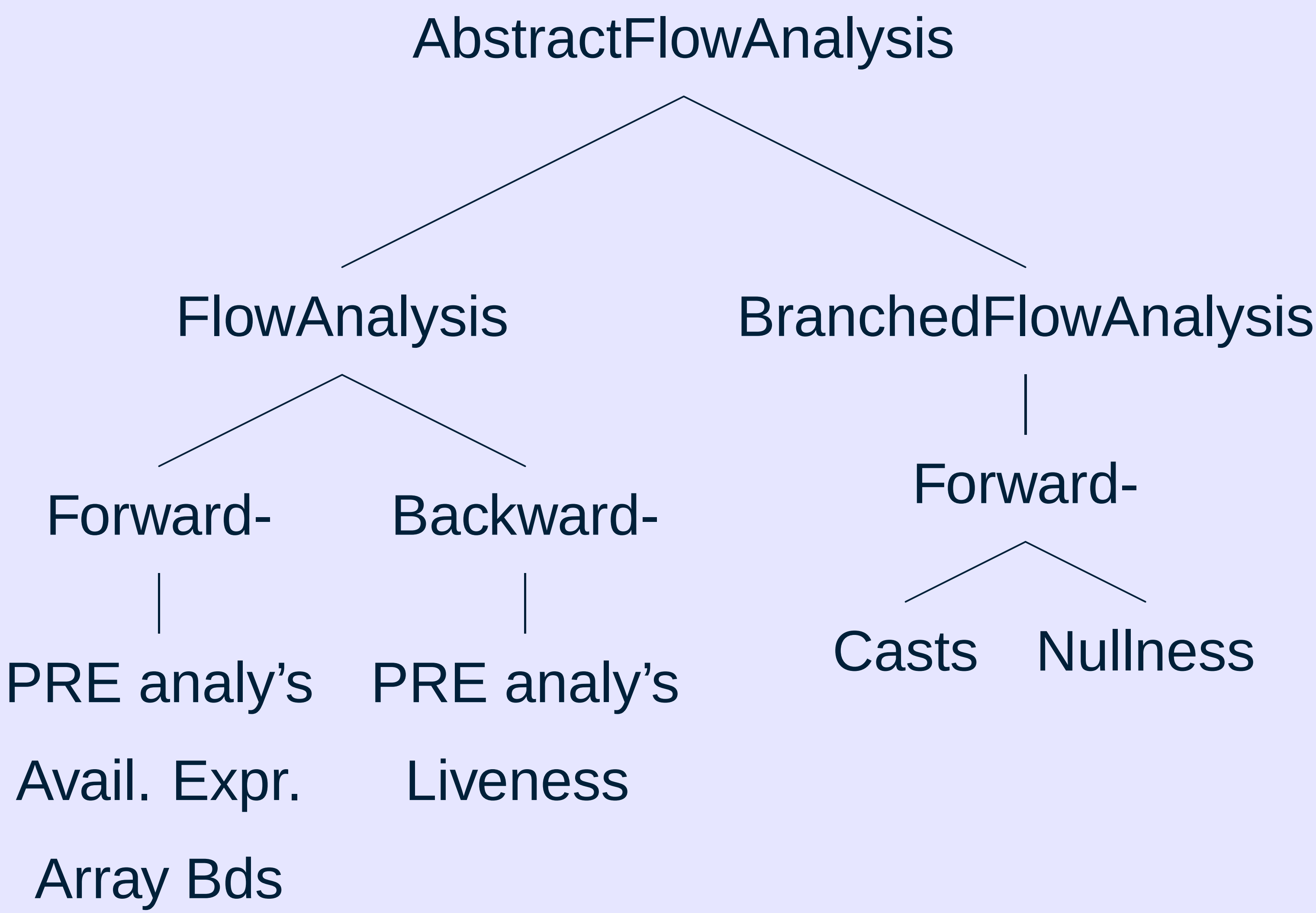
- impls of abstraction domains (flow sets)
  - standard abstractions trivial to implement;
- an implemented flow analysis namely,
  - `doAnalysis()` method: executes intraprocedural analyses on a CFG using a worklist algorithm.

Flow Analysis Hierarchy



oot.toolkits.scalar

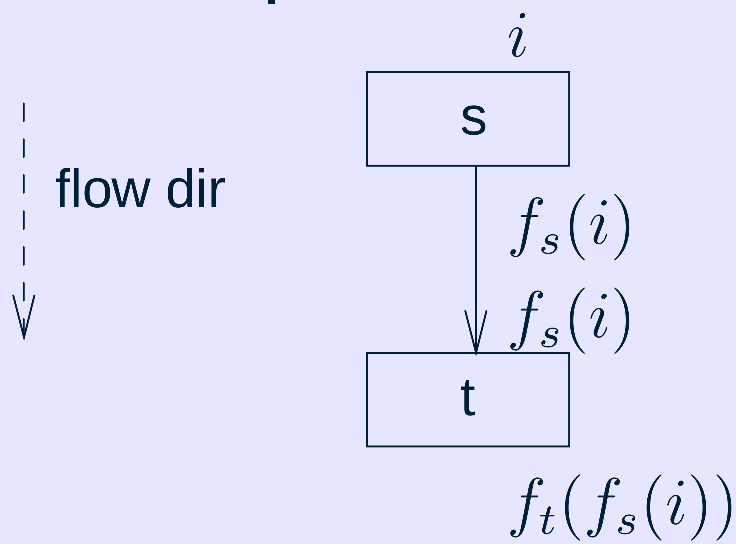
Soot Flow Analyses



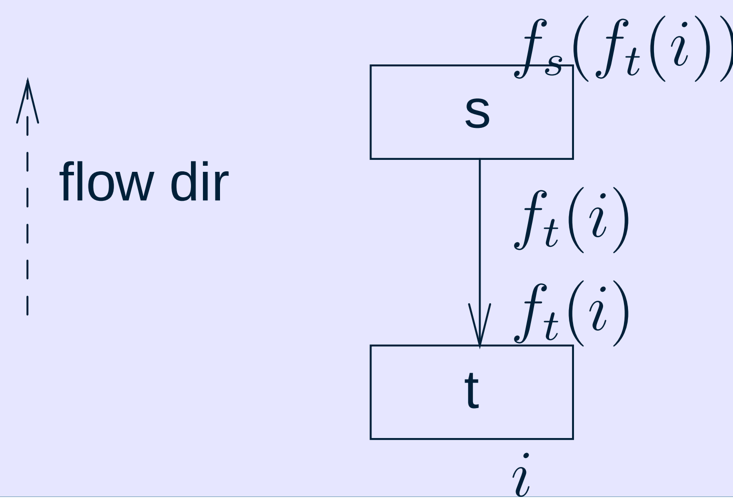
oot.toolkits.scalar

Backward vs. Forward Analyses

A forward analysis computes OUT from IN:



A backward analysis computes IN from OUT:



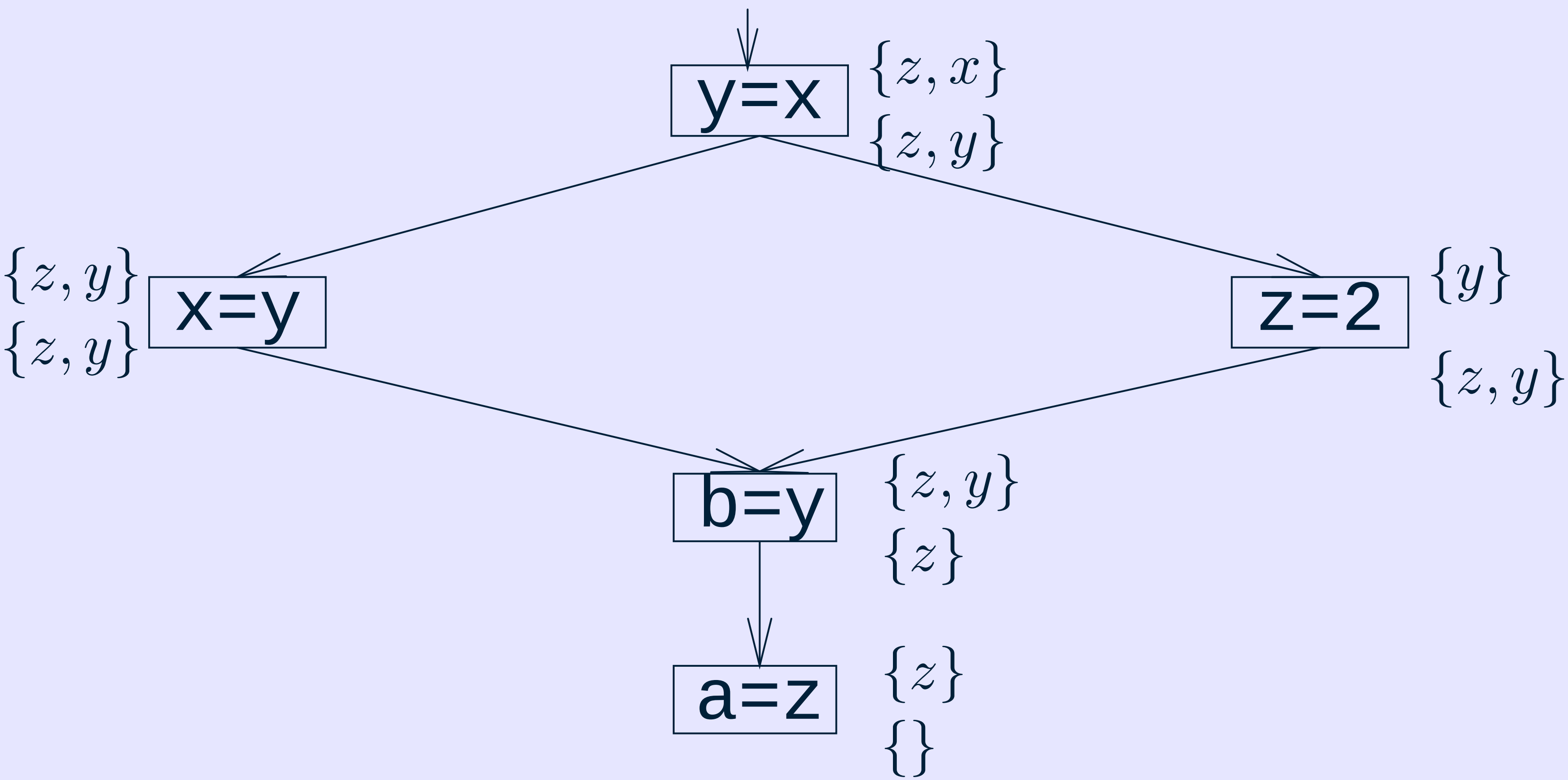
Outline: Soot Flow Analysis Examples

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()`)

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
```

oot.toolkits.scalar.BackwardFlowAnalysis

Step 2: Abstraction domain

Domain for Live Variables: sets of `Locals`

e.g. {`x, y, z`}

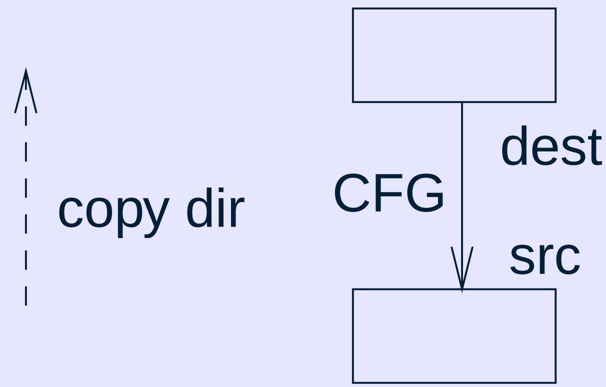
- Partial order is subset inclusion
- Merge operator is union

In Soot, we use the provided `ArraySparseSet` implementation of `FlowSet`.

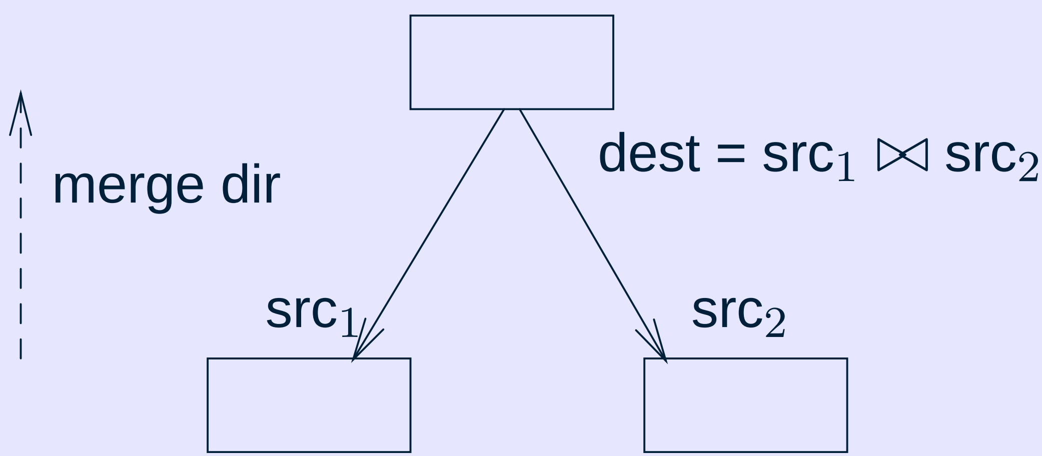
oot.toolkits.scalar.ArraySparseSet

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:

```
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`

// $c = a \cap b$

```
a.intersection(b, c);
```

// $c = a \cup b$

```
a.union(b,c);
```

// $d = \overline{c}$

```
c.complement(d);
```

// $d = d \cup \{\mathtt{v}\}$

```
d.add(v);
```

oot.toolkits.scalar.FlowSet

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 $\top$ to another `FlowSet`)

oot.toolkits.scalar.*Set

Step 2: copy() for live variables

```
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 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`:

```
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 * z`:

kill def x;

gen uses y, z.

How? Implement this method:

```
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:

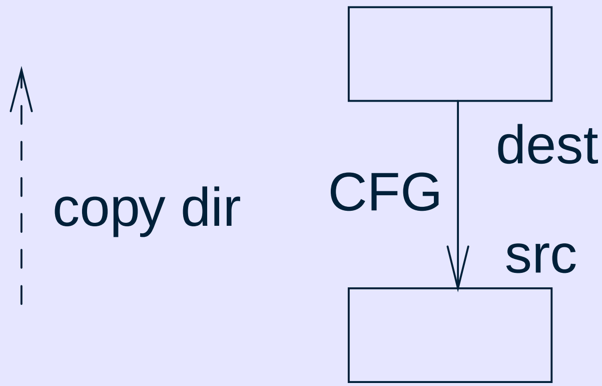
- casting `srcValue`, `destValue` to `FlowSet`.
- casting `u` to `Unit ut`.

In code:

```
FlowSet src = (FlowSet)srcValue,
        dest = (FlowSet)destValue;
Unit ut = (Unit)u;
```

Step 3: Copying

Need to copy `src` to `dest` to allow manipulation.

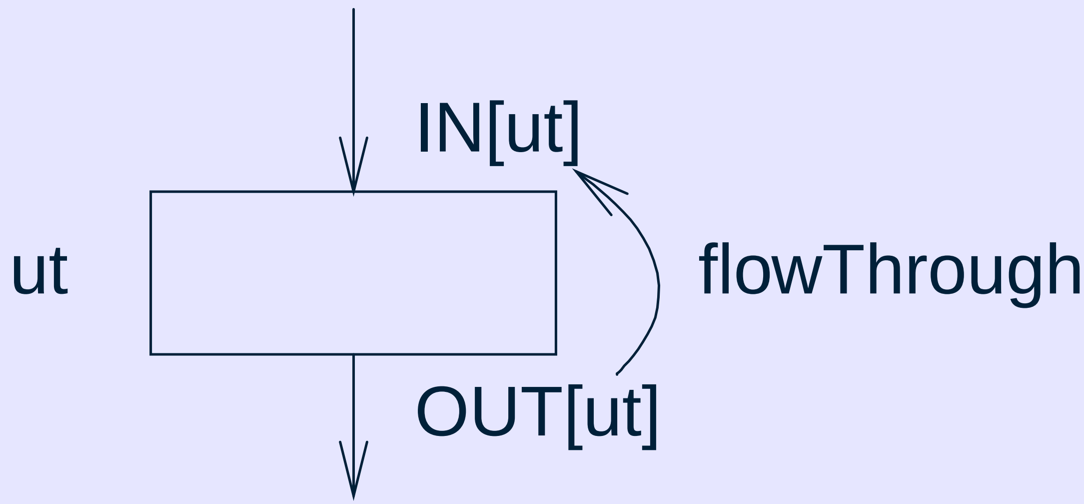



```
src.copy (dest);
```

Use `FlowSet` methods.

Step 3: Implementing `flowThrough`

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



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

`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.

ep 3: Implementing `flowThrough`: removing kills

```
// 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 );
}
```

tep 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
    ```
    Object newInitialFlow()
        {return new ArraySparseSet();}
    ```
  - Create initial sets for exit nodes
    ```
    Object entryInitialFlow()
        {return new ArraySparseSet();}
    ```

Want conservative initial value at exit nodes, optimistic value at all other nodes.

Step 5: Implement constructor

```
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.

Enjoy: Flow Analysis Results

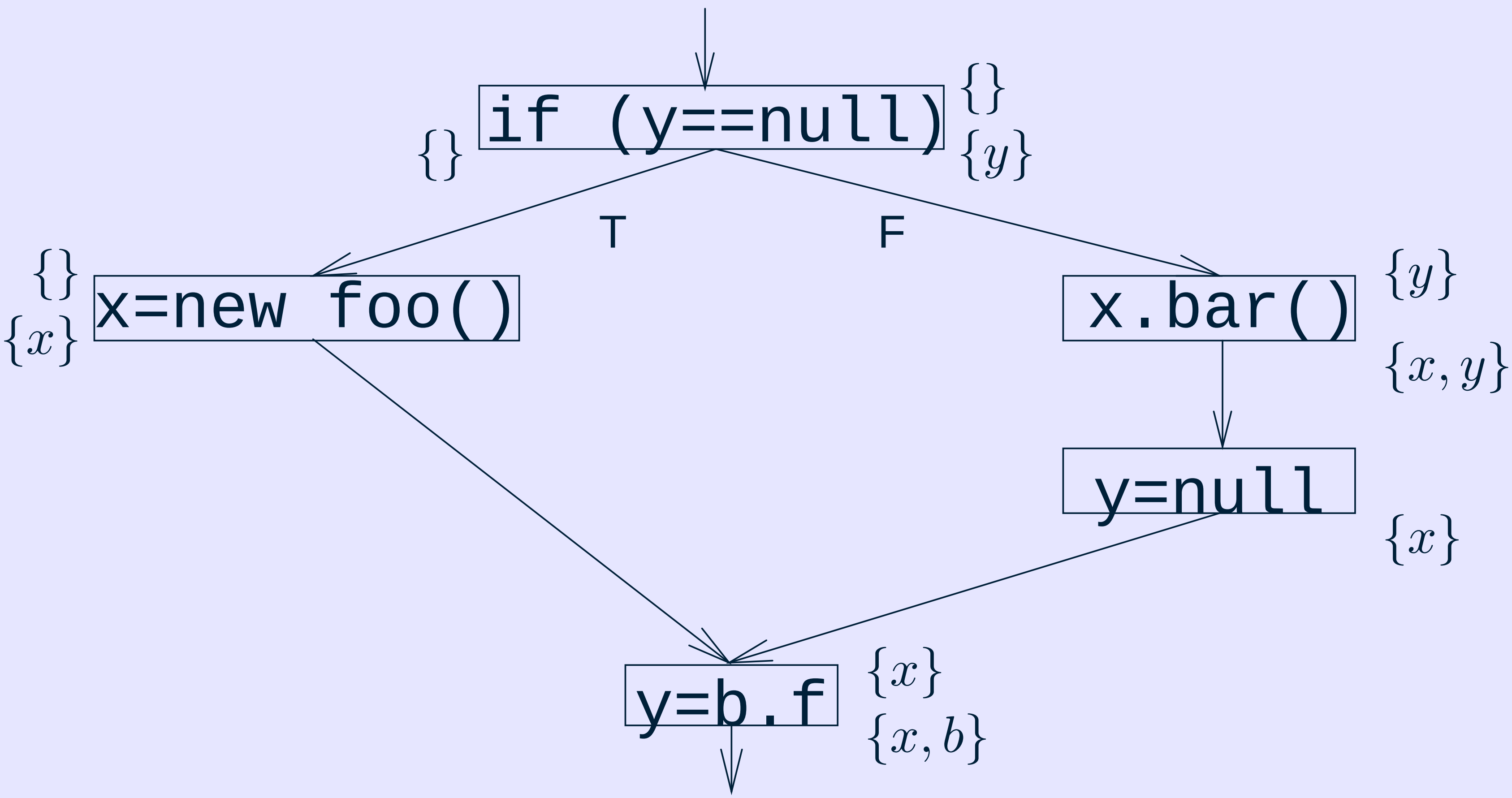
You can instantiate an analysis and collect results:

```
  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 `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()`)

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`.

```
class NullnessAnalysis
    extends ForwardBranchedFlowAnalysis {
```

oot.toolkits.scalar.ForwardBranchedFlowAnalysis

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. $\bot$, $\top$, `null`, non-`null` per-local.)

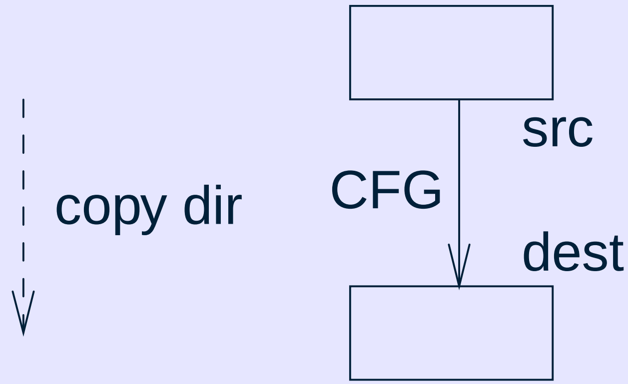
Again use `ArraySparseSet` to implement:

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

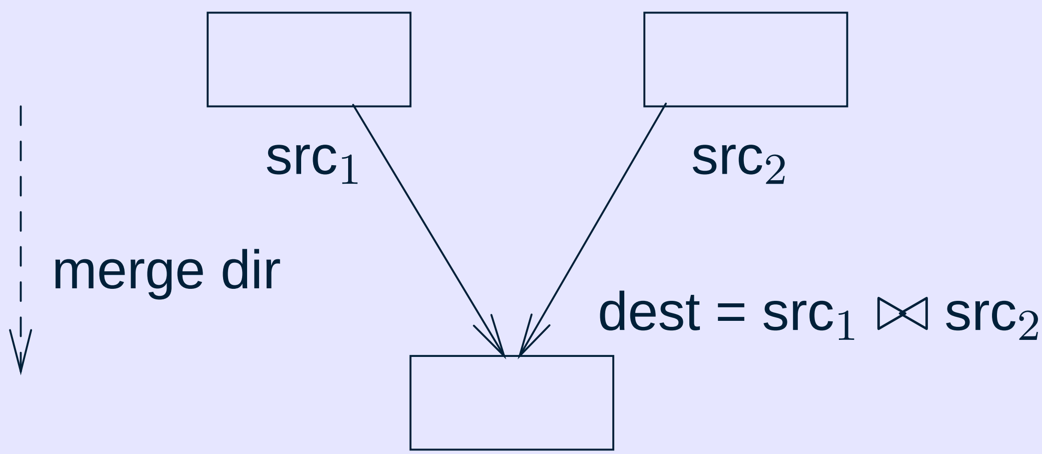
see soot.jimple.toolkits.annotation.nullcheck.BranchedRefVarsAnalysis

Implementing an Abstraction

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



`copy()` brings OUT set to predecessor's IN set.



`merge()` joins two OUT sets to make an IN set.

Step 2: `copy()` for nullness

Same as for live locals.

```
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()`:

```
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.

```
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.

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`;

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();`

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.

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` in `y = y.next`;
- Add gen set.
  `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.)

```
Object newInitialFlow() {
    { return fullSet.clone(); }
```

- Create entry sets (`emptySet` from constr.)

```
Object entryInitialFlow()
    { return emptySet.clone(); }
```

(To be created in constructor!)

Step 5: Constructor: Prologue

Create auxiliary objects.

```
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:

```
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.

Step 5: Constructor: Doing work

Don't forget to call `doAnalysis()`!

```
        ...

        doAnalysis();
    }
}
```

Enjoy: Branched Flow Analysis Results

To instantiate a branched analysis & collect results:

```
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.

$\Rightarrow$ Group them in a `Pack`.

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

```
Pack jtp = G.v().PackManager().
                    getPack("jtp");
jtp.add(new Transform("jtp.nt",
          new NullTransformer()));
jtp.add(new Transform("jtp.nac",
          new NullnessAnalysisColorer()));
```

oot.Pack

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()`

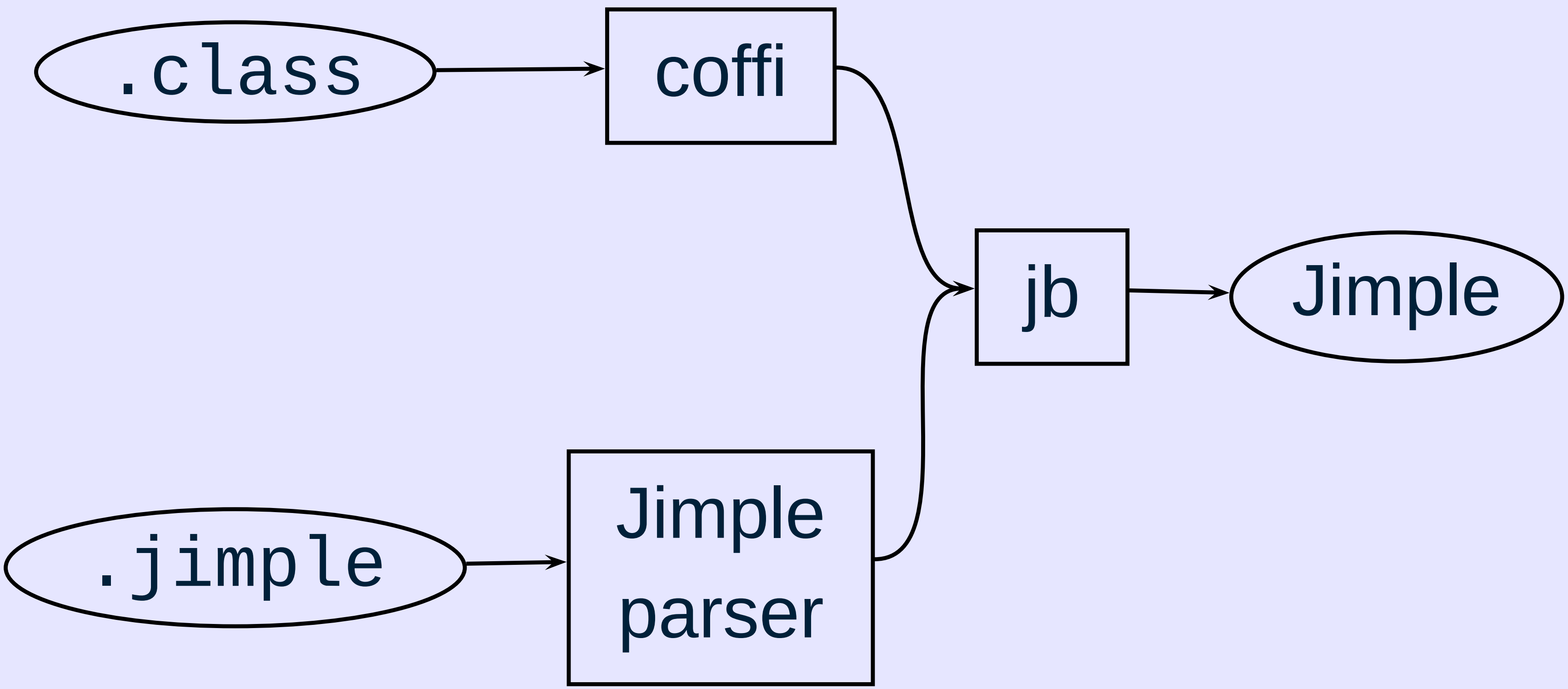
You can do any or all of these yourself:

- `Options.v()` contains setter methods for all options

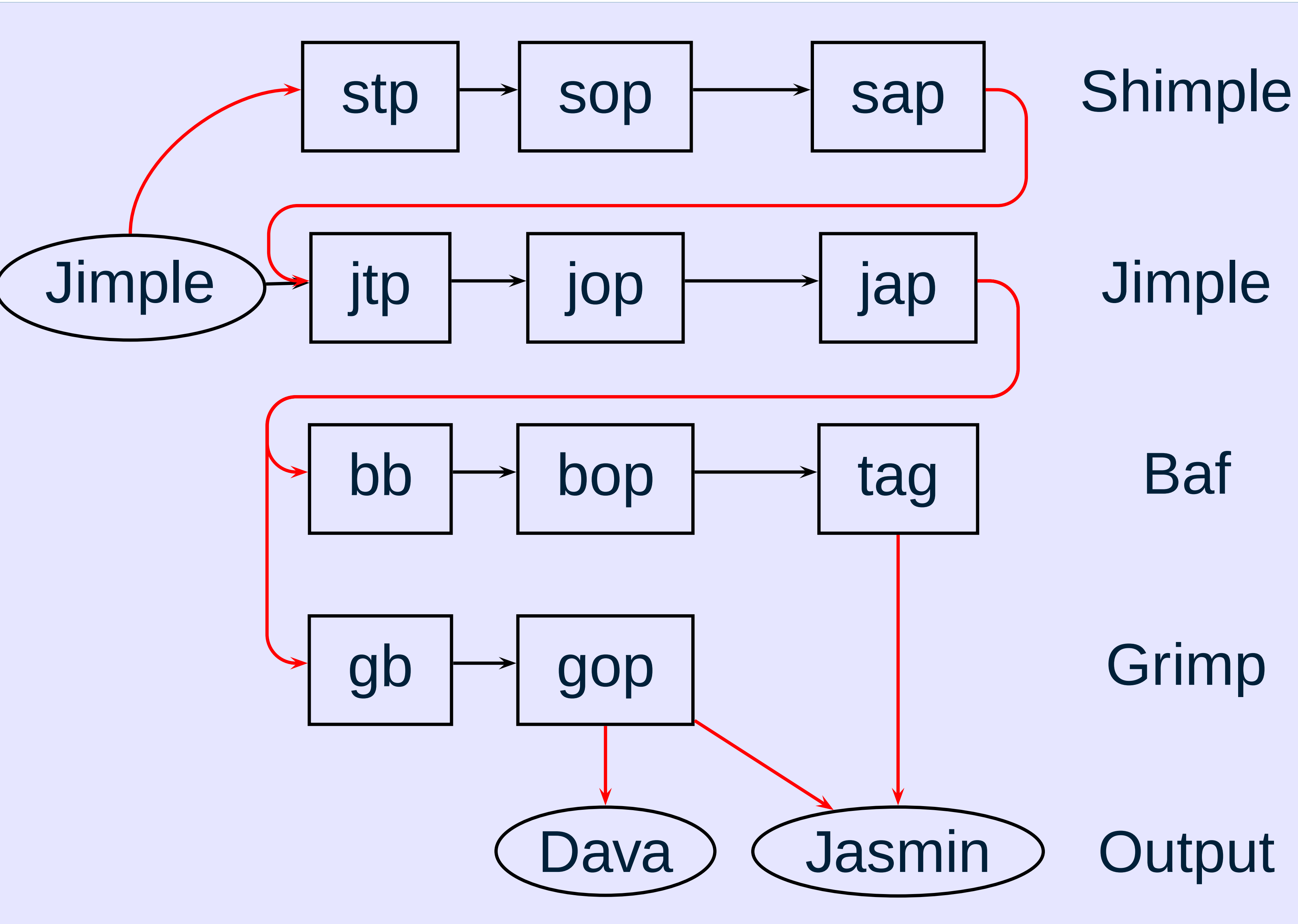
Running Soot more than once

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

Generating Jimple



Intra-procedural packs



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

he p is sometimes silent.

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

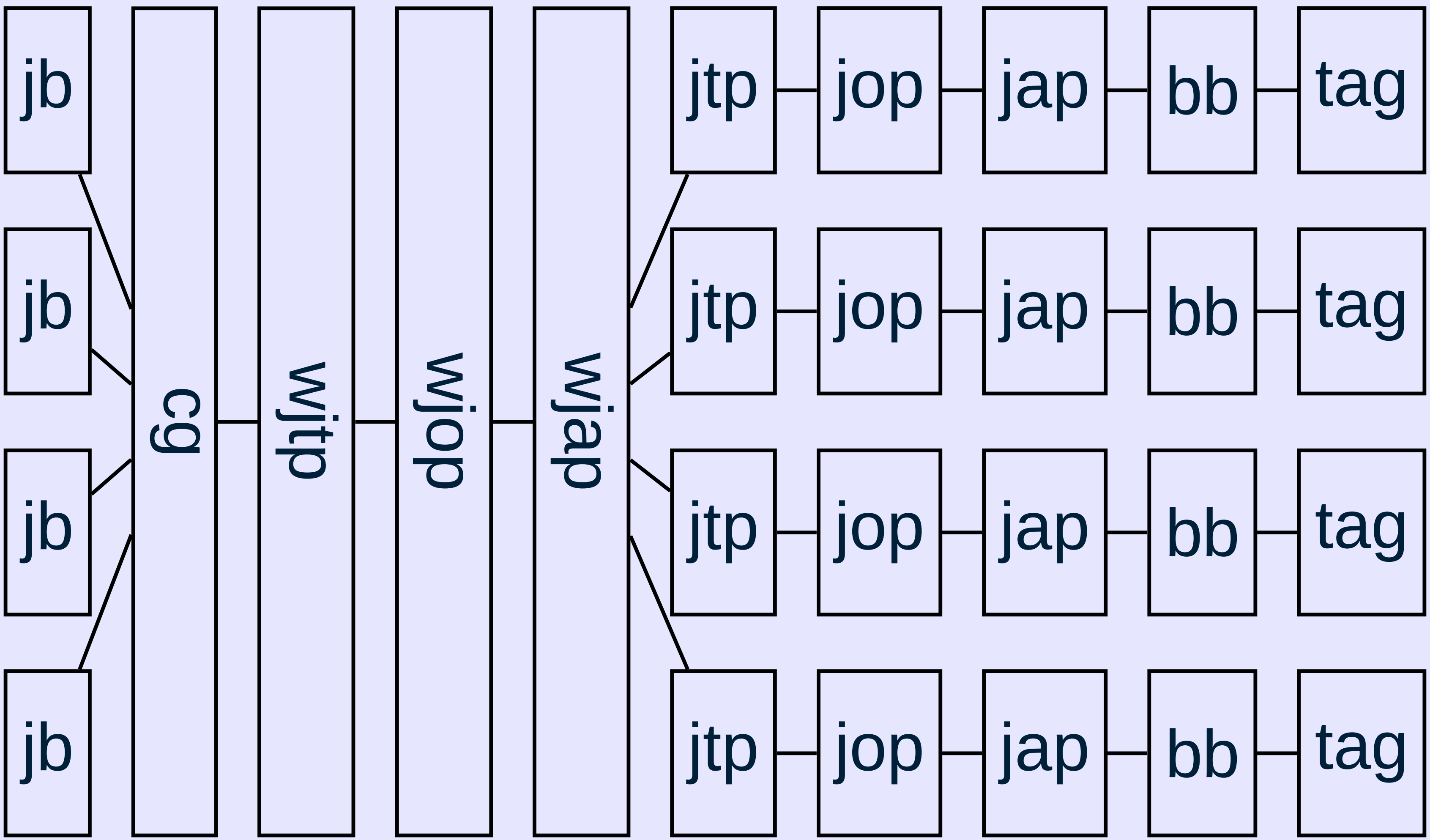
wjop performs whole-program optimizations

- static inlining
- static method binding

wjap generates annotations using whole-program analyses

- rectangular array analysis

Soot phases


jb
jb
jb
jb
cg
wjtp
wjop
wjap
jtp
jop
jap
bb
tag
jtp
jop
jap
bb
tag
jtp
jop
jap
bb
tag
jtp
jop
jap
bb
tag

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

source m.	source stmt.	target m.	kind
•	•	•	VIRTUAL

```
foo() {
   o.bar();
}
```

```
bar() {
   /* */
}
```

Edge Kinds

```
/** Due to explicit invokestatic instruction. */
public static final int STATIC = 1;
/** 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;
```

oot.jimple.toolkits.callgraph.Edge

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

main()	o.foo();	C1.foo()	VIRTUAL
main()	o.goo();	C1.goo()	VIRTUAL
main()	o.goo();	C2.goo()	VIRTUAL
bar()	o.foo();	C2.foo()	VIRTUAL

oot.jimple.toolkits.callgraph.CallGraph

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

main()	o.foo();	C1.foo()	VIRTUAL
main()	o.goo();	C1.goo()	VIRTUAL
main()	o.goo();	C2.goo()	VIRTUAL
bar()	o.foo();	C1.foo()	VIRTUAL

oot.jimple.toolkits.callgraph.CallGraph

Adapters

- Adapters make an iterator over edges into an iterator over

  `Sources` source methods

  `Units` source statements

  `Targets` target methods

src_1	$stmt_1$	tgt_1	$kind_1$
src_2	$stmt_2$	tgt_2	$kind_2$
src_3	$stmt_3$	tgt_3	$kind_3$

src_1
src_2
src_3

oot.jimple.toolkits.callgraph.{Sources,Units,Targets}

Code Example

```
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

Code Example

```
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
}
```

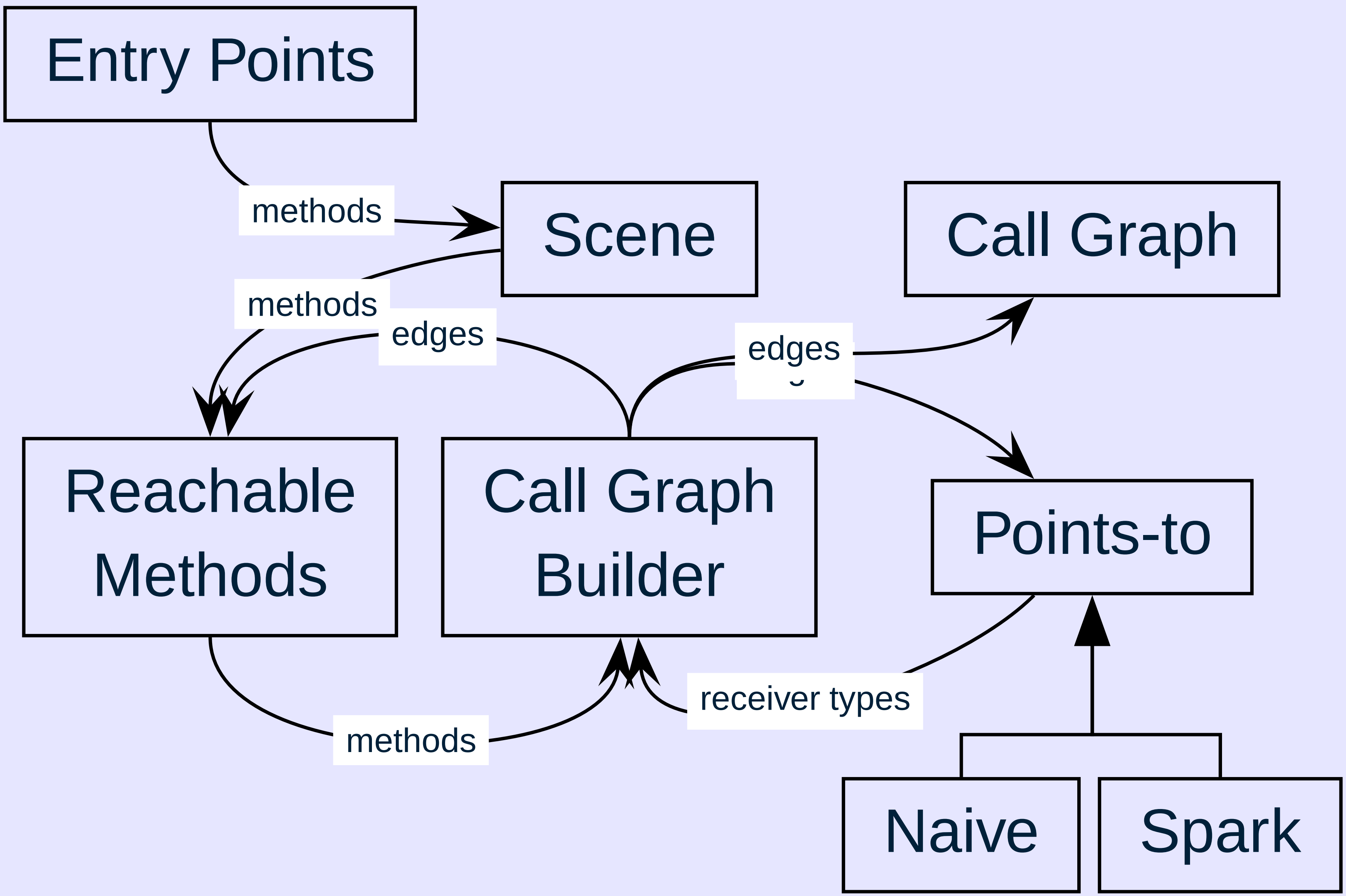
Transitive Targets

- `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

Implementation Big Picture



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`

oot.PointsToAnalysis

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)

oot.PointsToSet

If I want to know...

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

```
o.m(...)
```

```
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 =
    pa.reachingObjects( x );
PointsToSet yset =
    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

oot.SideEffectTester

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

oot.jimple.NaiveSideEffectTester

oot.jimple.toolkits.pointer.PASideEffectTester

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

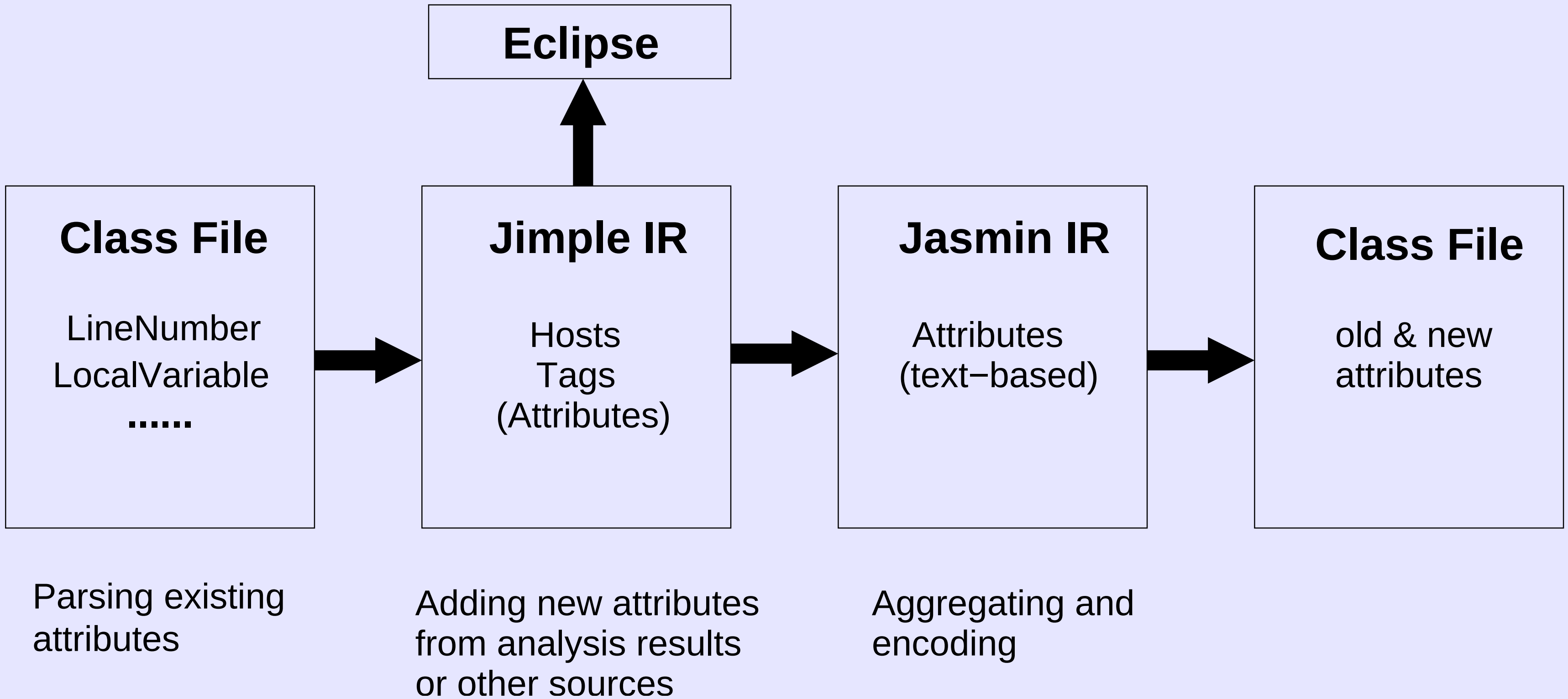
Attribute format

The VM spec defines the format of attributes:

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

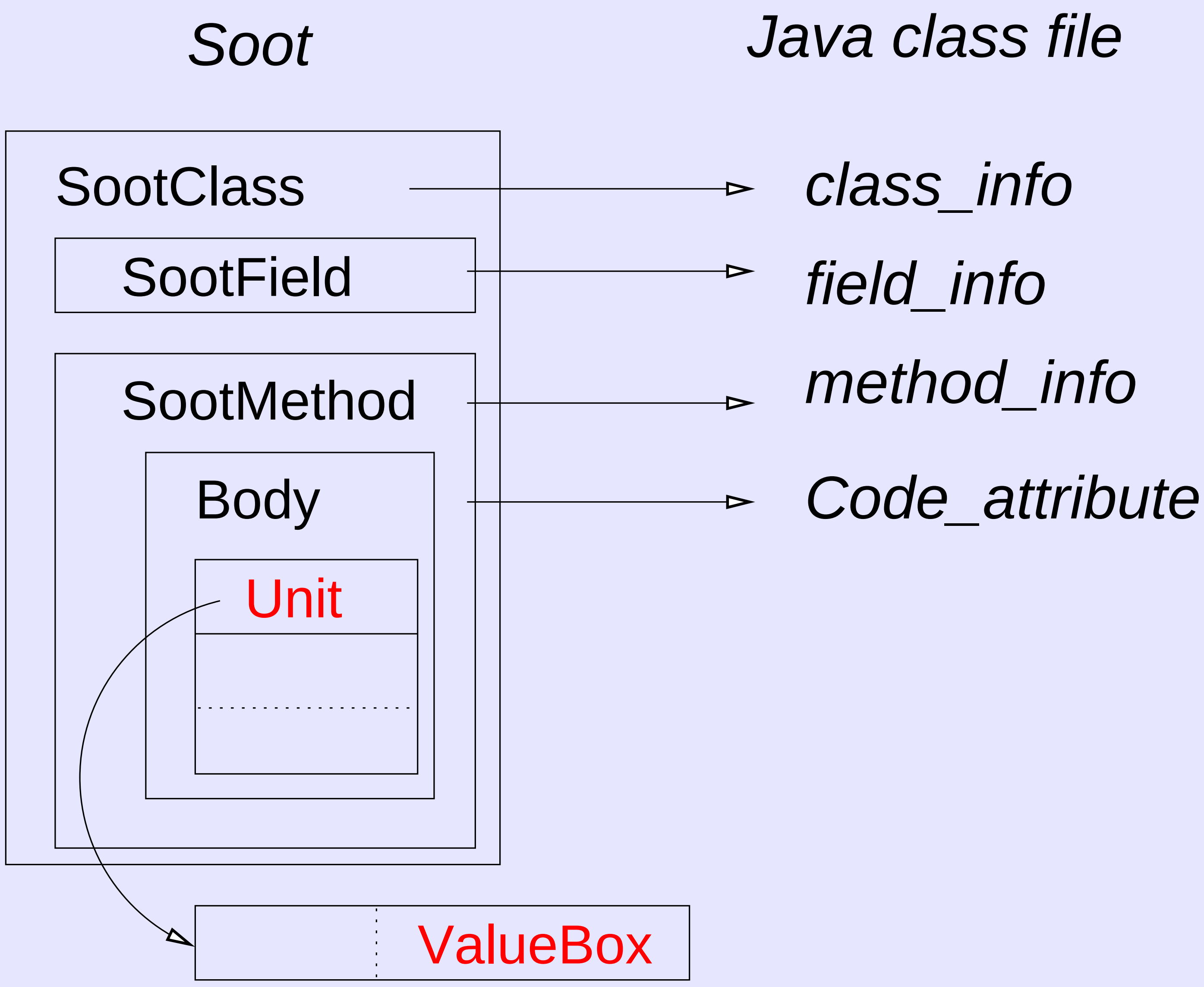
- `attribute_name_index`, the index of the attribute's name in the class files' *Constant Pool*
- `attribute_length`, the length of the attribute's data
- `info`, an array of raw attribute data

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



Hosts

Hosts are objects that can hold Tags:

```
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

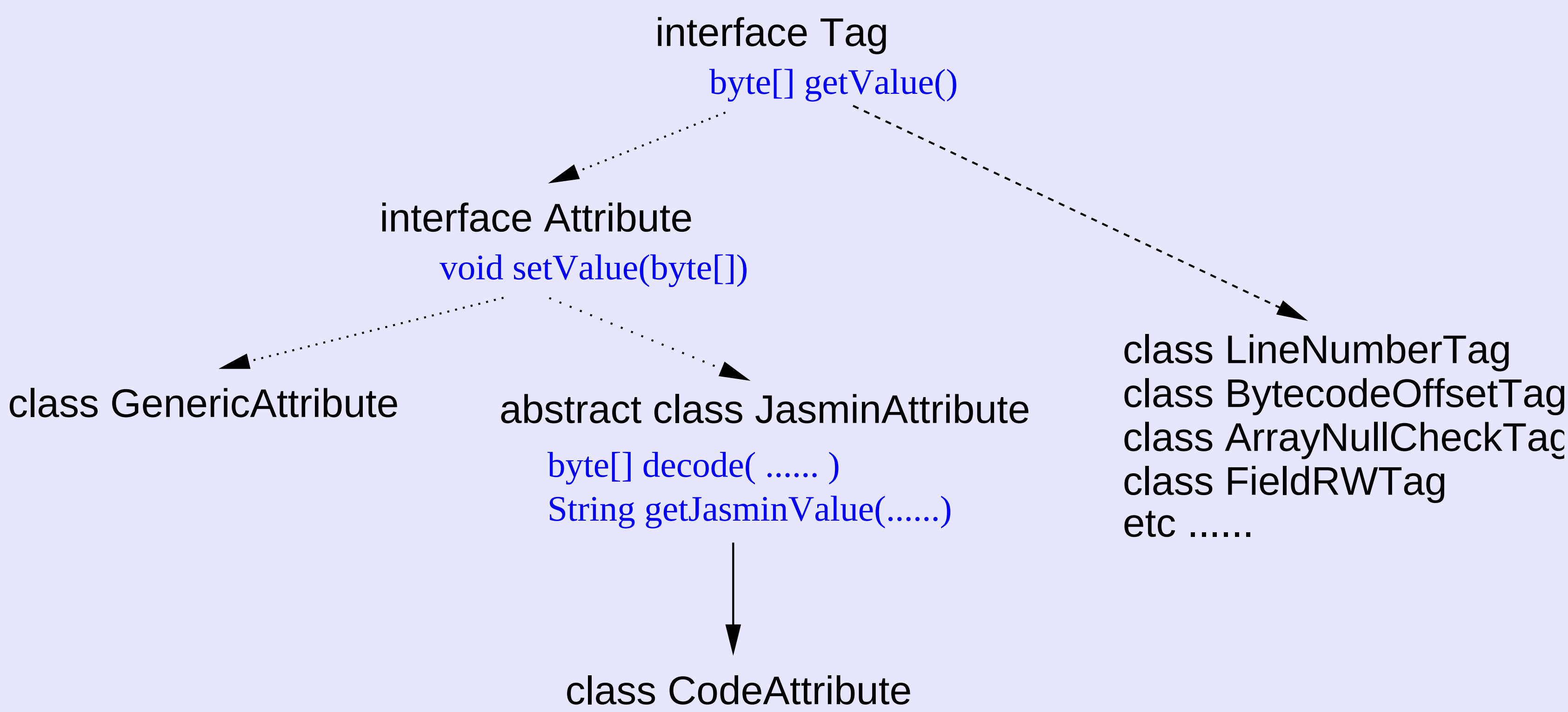
Tags

Tags are objects that can be attached to Hosts:

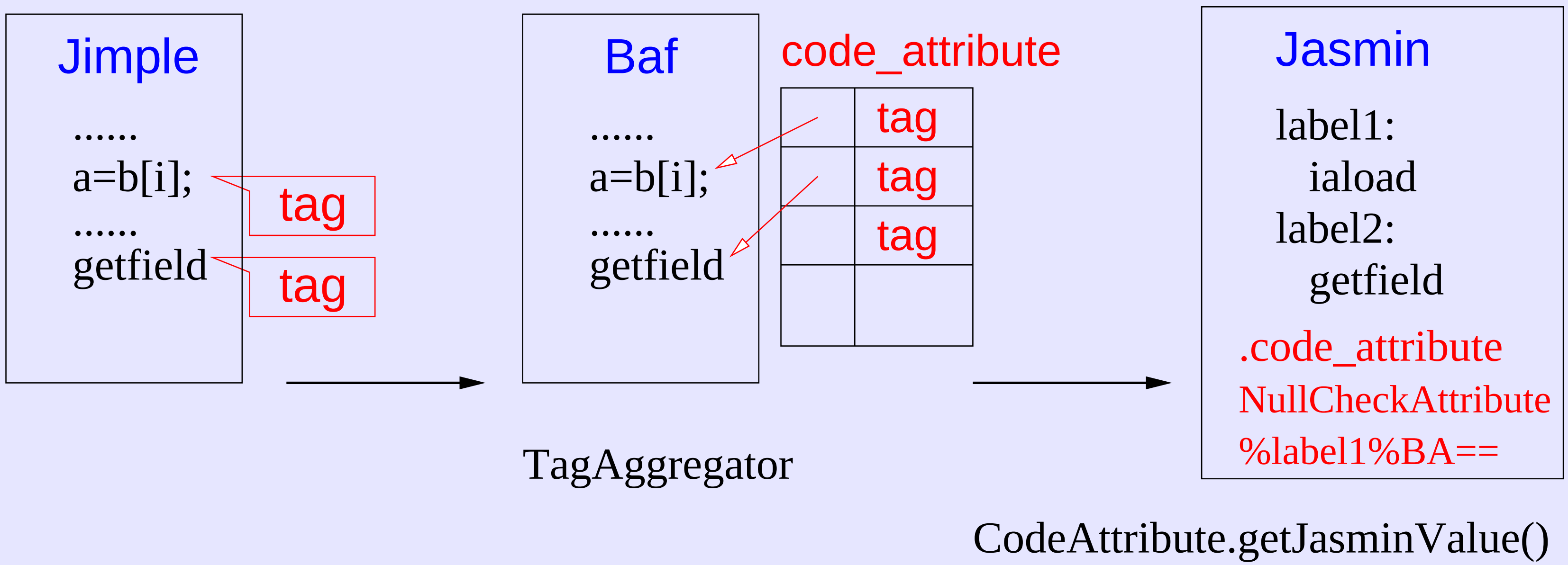
```
package soot.tagkit;
public interface Tag {
  public String getName ();
  public byte[] getValue ()
    throws AttributeValueException;
  public String toString();
}
```

- Attribute attached to class file structures (class, field, method)
- Generic tags attached to *Units* or *ValueBoxes*

Tag Hierarchy



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

Choosing an Aggregator

- One Jimple statement may translate to multiple bytecode instructions

Jimple	Bytecode

`x = y.f`	`load y`
	`getfield f`
	`store x`

- 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 . . .

TagAggregator

```
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, ... ) {
    ......
  }
}
```

ImportantTagAggregator

```
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]");
  }
}
```

Example: nullness attribute

Step 2: attach tags to units after analysis

```
boolean needCheck;
s.addTag(new NullCheckTag(needCheck));
```

Example: nullness attribute

Step 3: create a NullTagAggregator

```
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";
    }
}
```

Code attribute format

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

```
String getJasminValue(Map instToLabel);
```

e.g. "NullCheckAttribute":

```
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.

```
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.

oot.tagkit.StringTag

Color Tags

- ColorTags attach a color to a Host.

```
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.

oot.tagkit.ColorTag

Link Tags

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

```
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 a another part of the code when clicked in the Jimple Editor.

oot.tagkit.LinkTag

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.

Homework

- 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

Introduction to Soot (1.x): Raja's thesis, CASCON 99, CC 2000, SAS 2000

Initial design of attributes: CC 2001

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

Decompiling: Jerome's thesis, WCRE 2001, CC 2002

Points-to analysis: Ondřej's thesis, CC 2003, PLDI 2003 (BDD-based)