Analyzing Java Programs with Soot

Bruno Dufour

based on material by:

Eric Bodden,Laurie Hendren, Patrick Lam, Jennifer Lhoták, Ondřej Lhoták and Feng Qian

McGill University

`http://www.sable.mcgill.ca/soot/`

What is Soot?

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

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

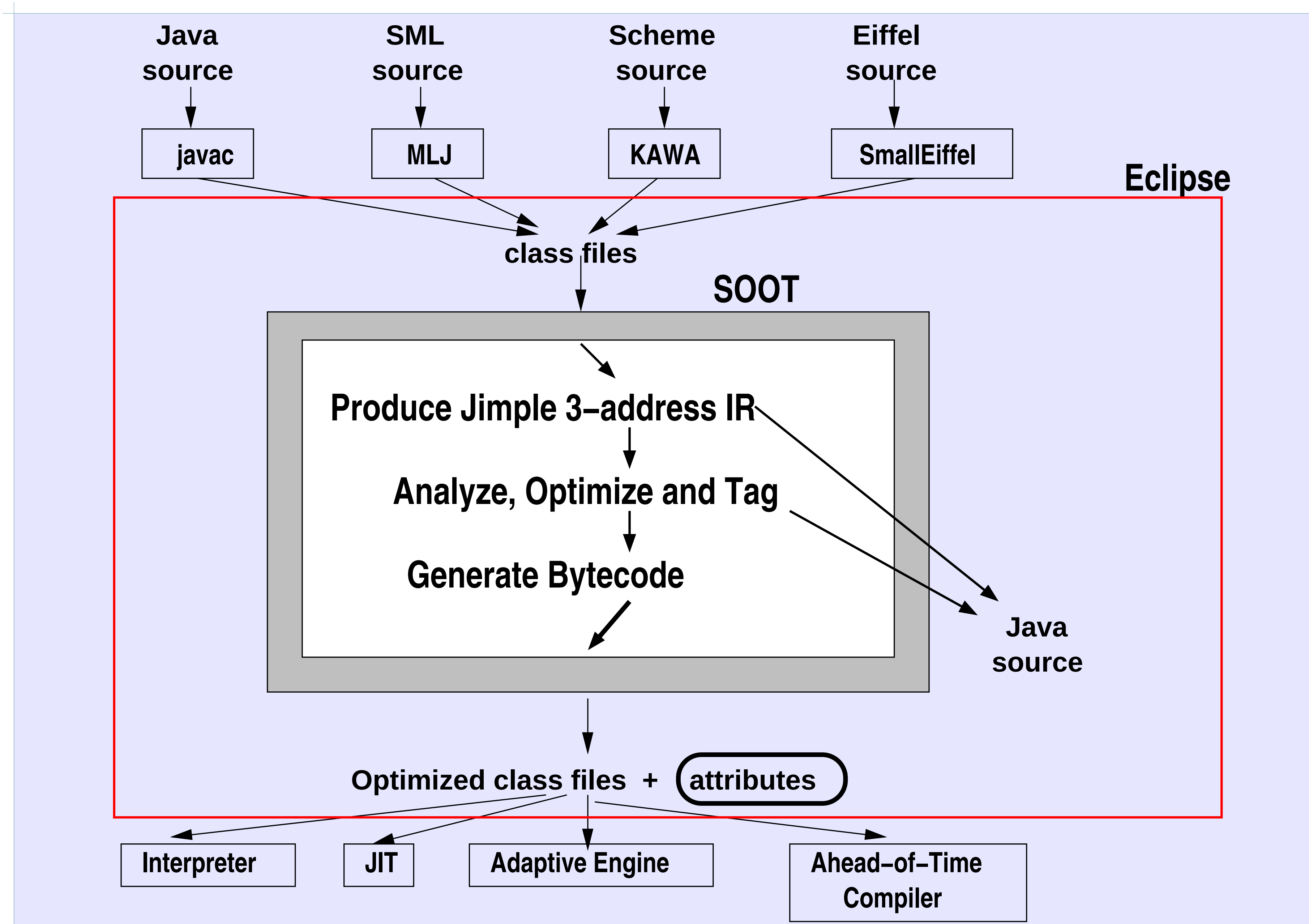
- Started in 1996-97 with the development of `coffi` by Clark Verbrugge and some first prototypes of `Jimple` IR by Clark and Raja Vallée-Rai.
- First publicly-available versions of Soot 1.x were associated with Raja's M.Sc. thesis
- New contributions and releases have been added by many graduate students 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 were released - June 2003 - JIT for PLDI 2003.
- Soot 2.3.0: Java 5 support
- Soot 2.4.0: partial Reflection support
- This tutorial is based on Soot 2.4.0.

Soot Overview



Soot IRs

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

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

Shimple: is a SSA-version of Jimple

Grimp: is like Jimple, but with expressions agGRegated

Dava: structured representation used for Decompiling Java

Jimple

Jimple is:

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

Raja'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

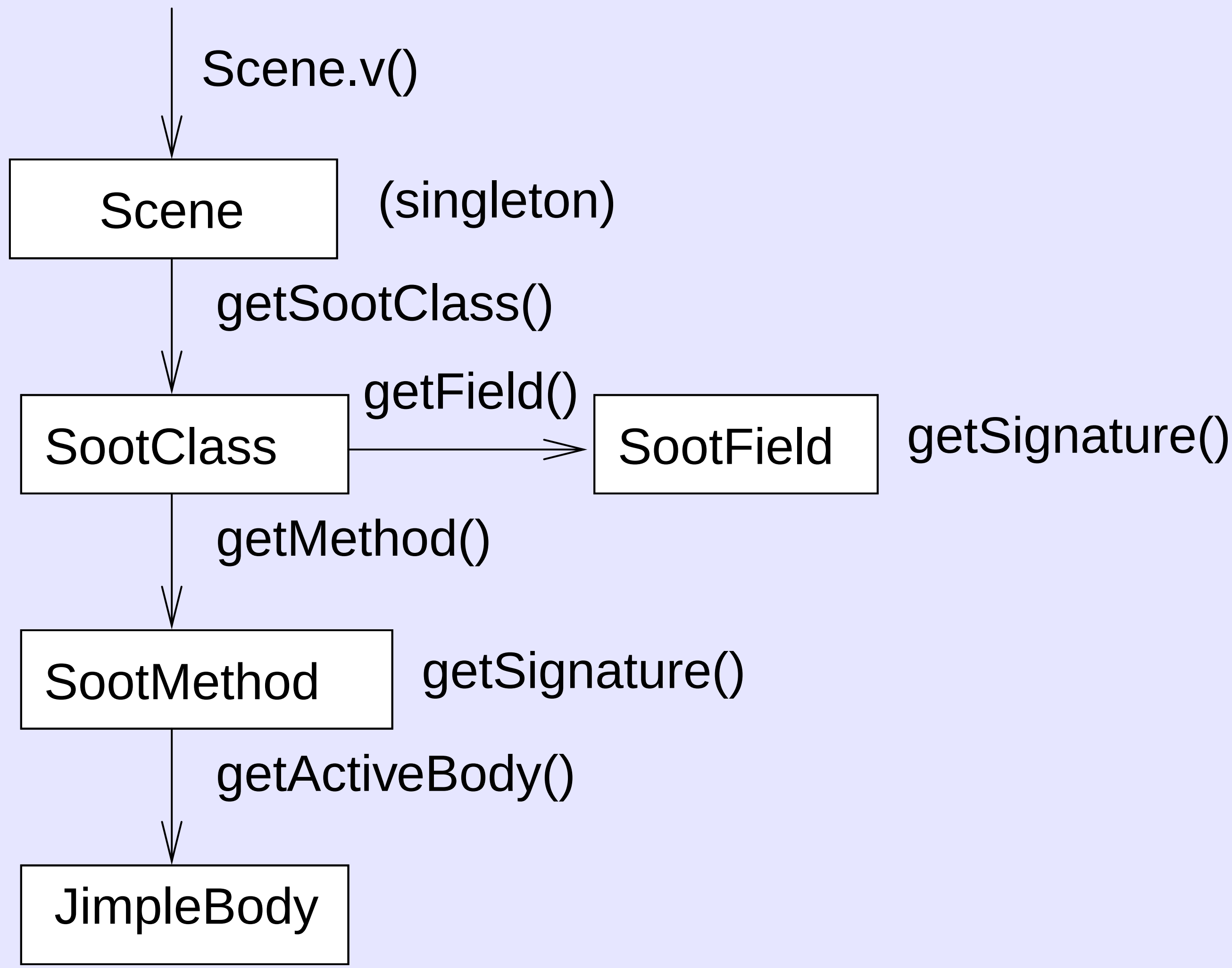
- These transformations are relatively hard to design so that they produce correct, useful and efficient code.
- Worth the price, we do want a 3-addr typed IR.

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

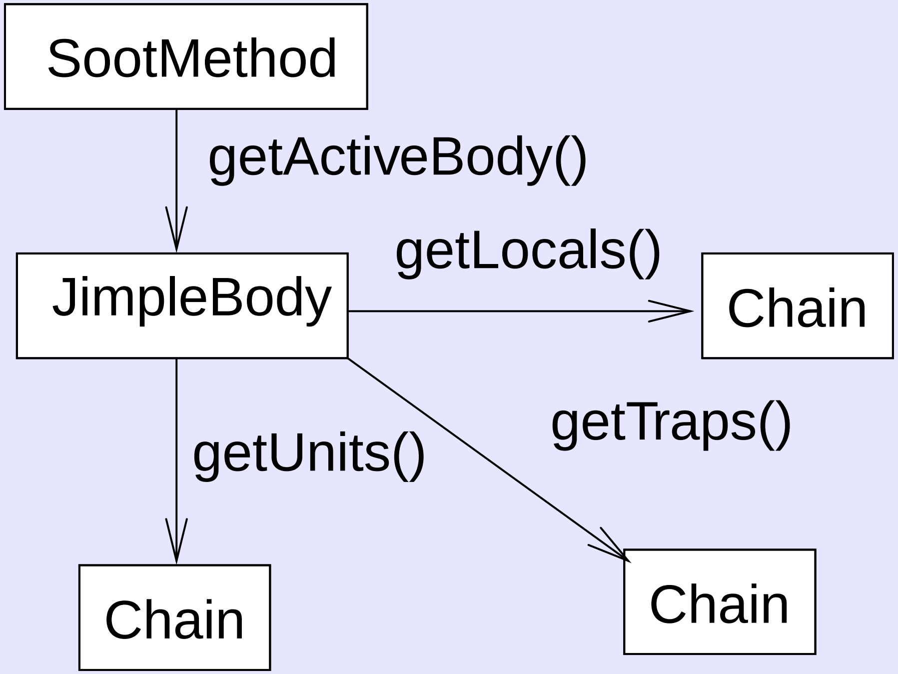
Soot Data Structure Basics

- 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.

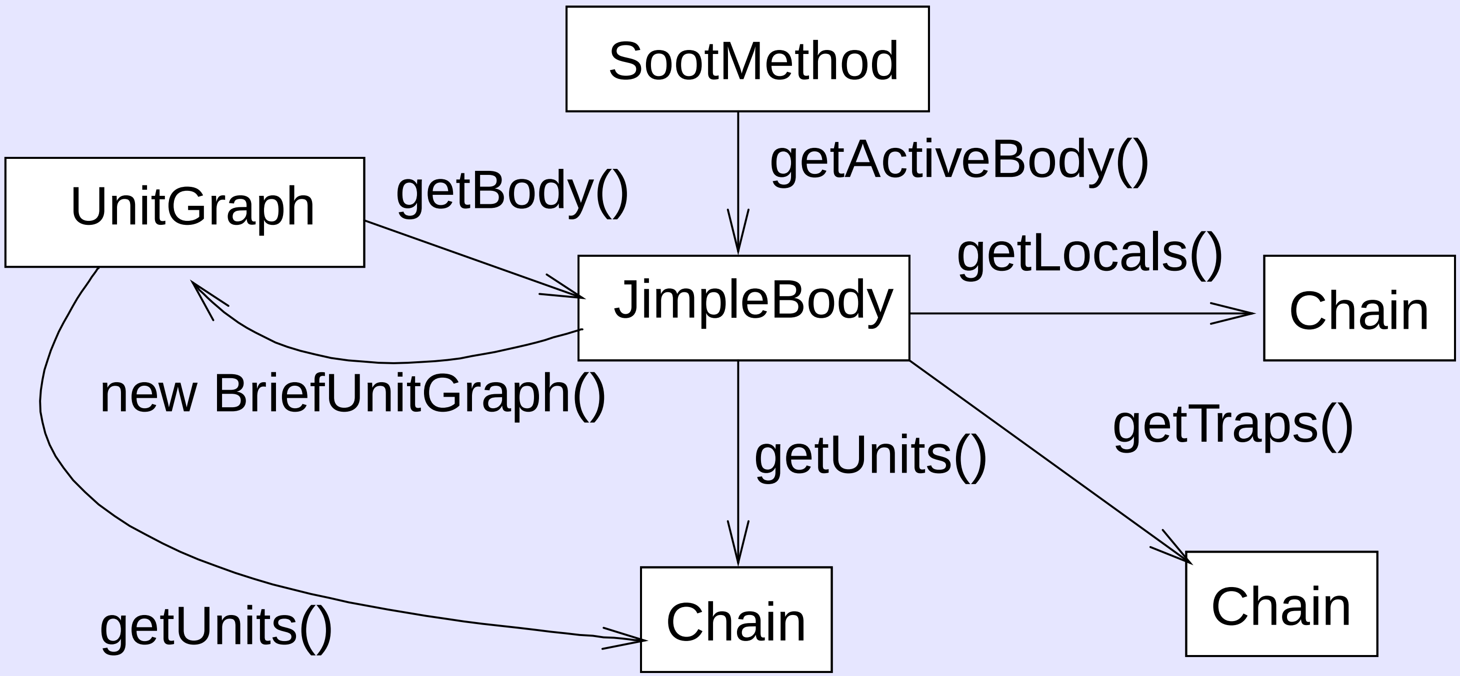
Soot Classes



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!

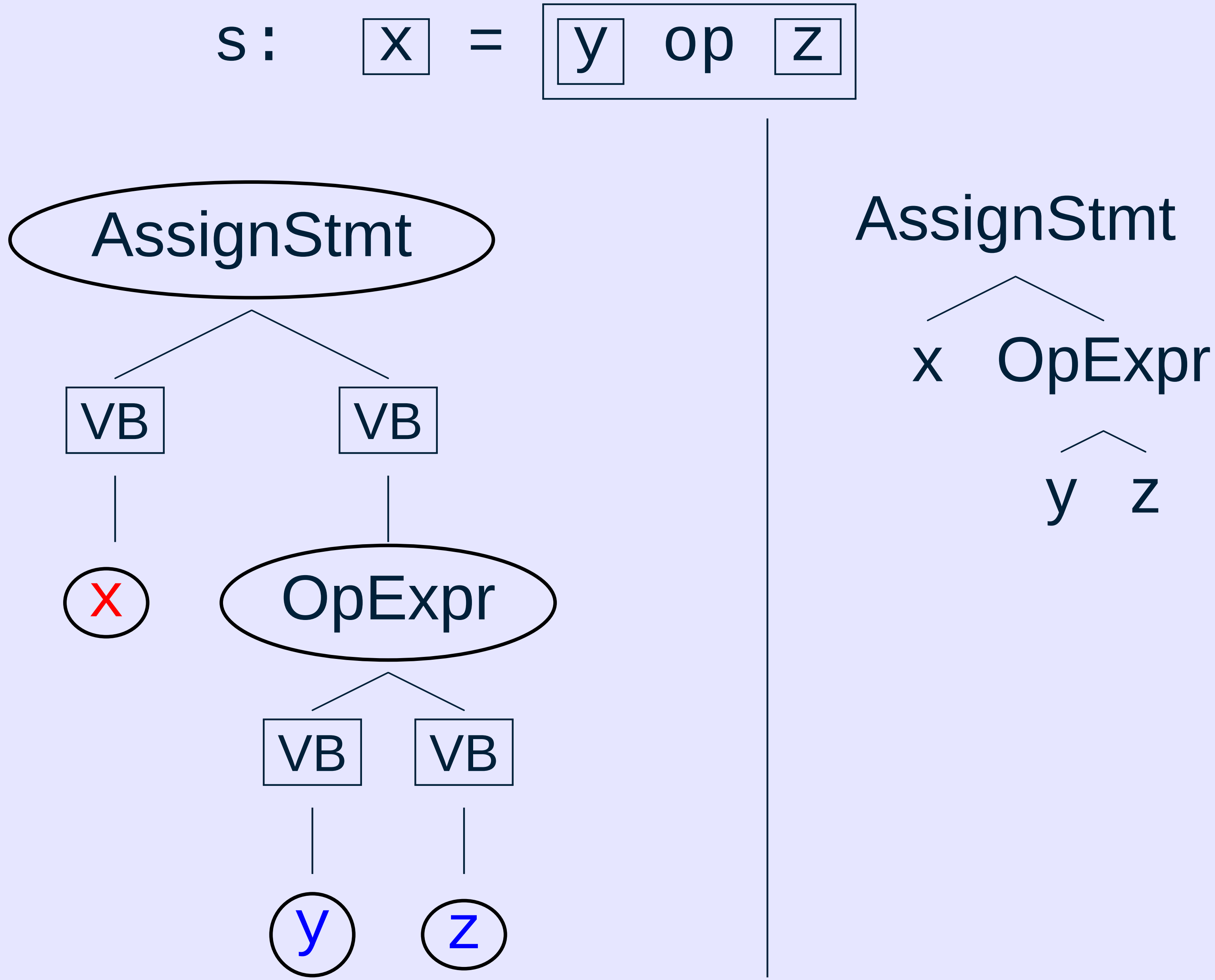
Accessing data:

- `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.

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

- `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`.

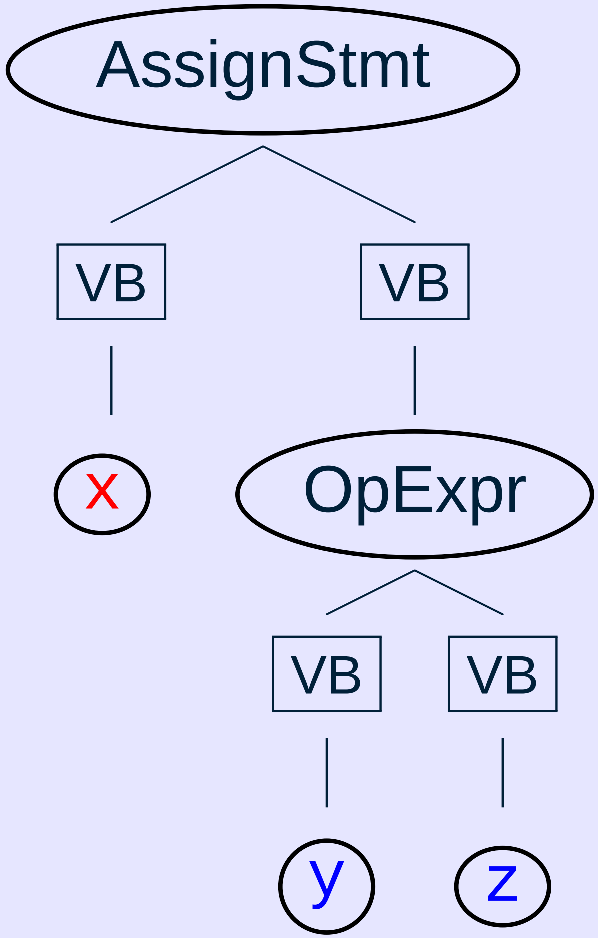
ut: $\boxed{\texttt{x}}$ = $\boxed{\boxed{\texttt{y}}\ \texttt{op}\ \boxed{\texttt{z}}}$;

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

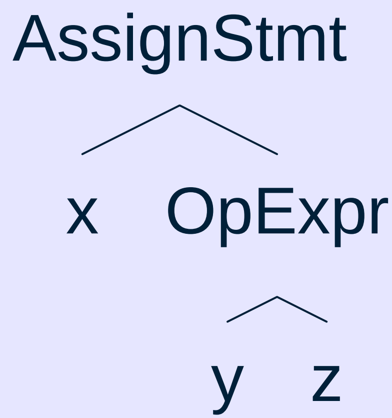
(`List` containing 3 `ValueBoxes`, 2 containing `Locals` & 1 `Expr`)

Why Boxes?

Change all instances of y to `1`:



`setValue()`



??

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()`

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:

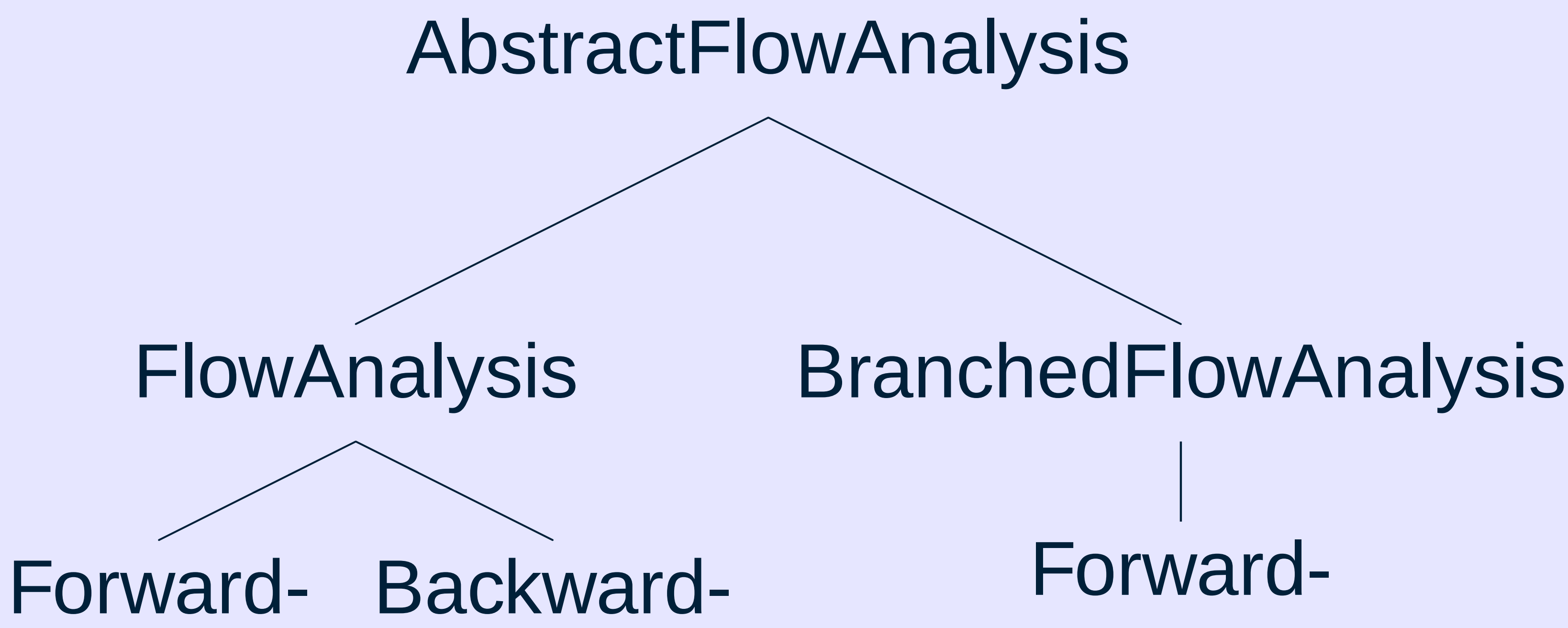
1. Subclass `*FlowAnalysis`
2. Implement abstraction: `merge()`, `copy()`
3. Implement flow function `flowThrough()`
4. Implement initial values:
   `newInitialFlow()` and
   `entryInitialFlow()`
5. Implement constructor
   (it must call `doAnalysis()`)

HOWTO: Soot Flow Analysis II

Soot provides you with:

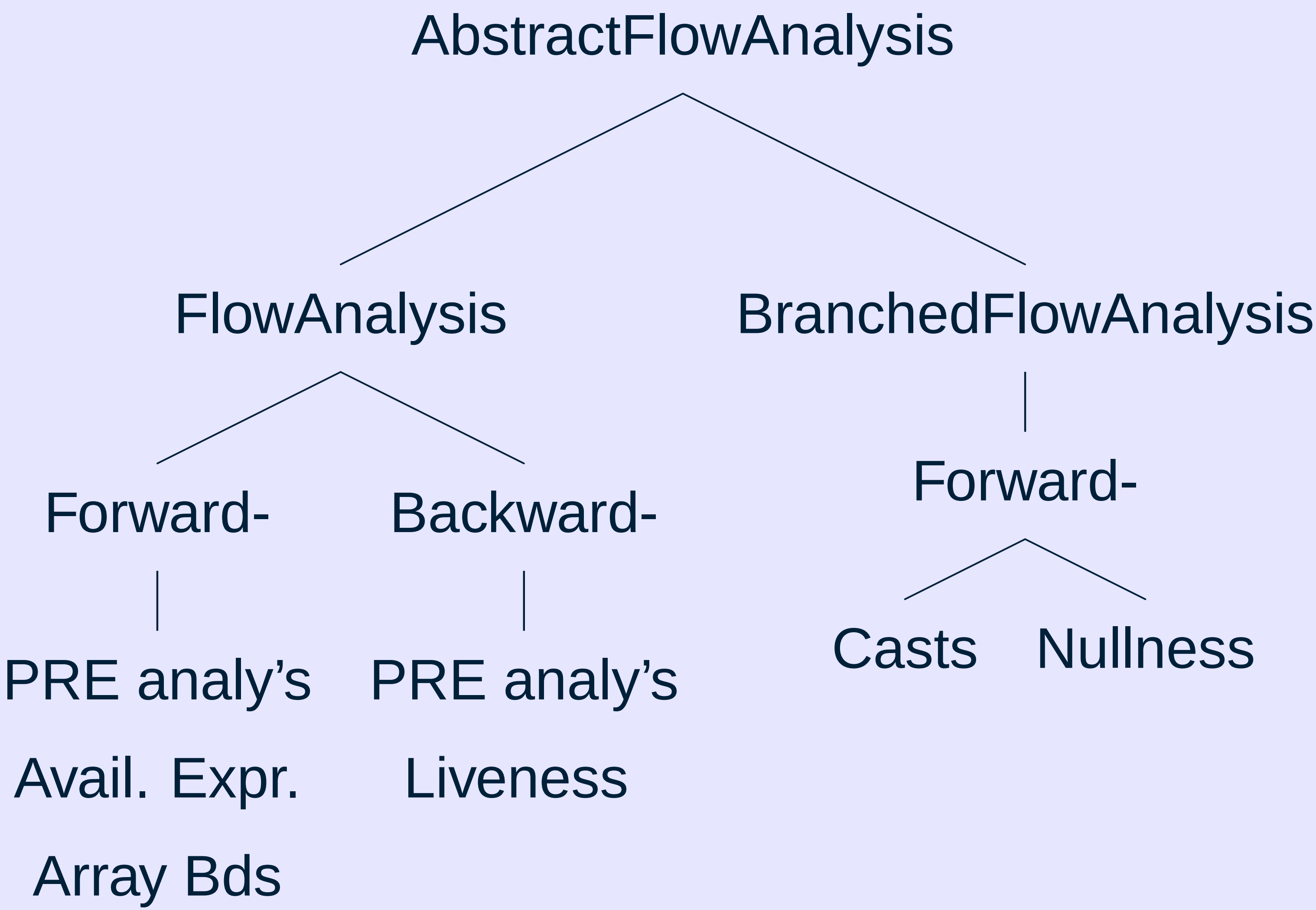
- impls of abstraction domains (flow sets)
  - standard abstractions trivial to implement;
- an implemented flow analysis namely,
  - `doAnalysis()` method: executes intraprocedural analyses on a CFG using a worklist algorithm.

Flow Analysis Hierarchy



soot.toolkits.scalar

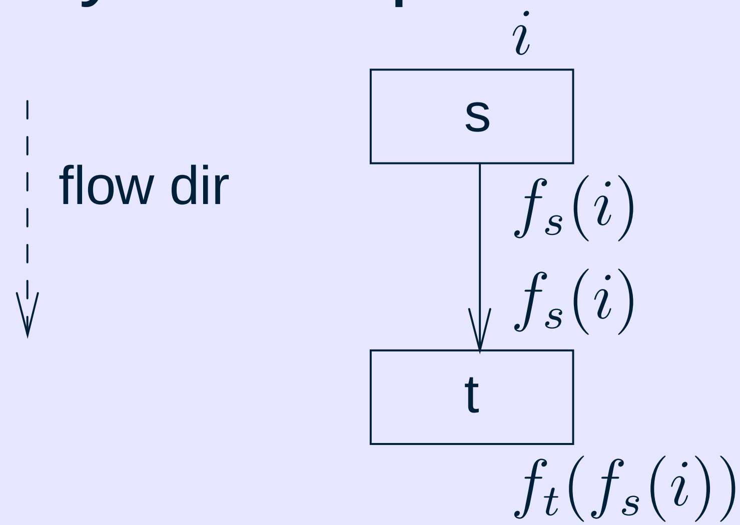
Soot Flow Analyses



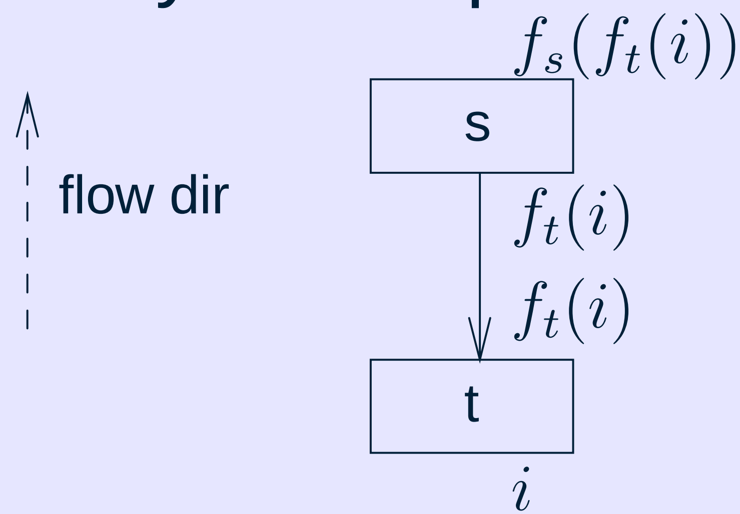
soot.toolkits.scalar

Backward vs. Forward Analyses

A forward analysis computes OUT from IN:



A backward analysis computes IN from OUT:



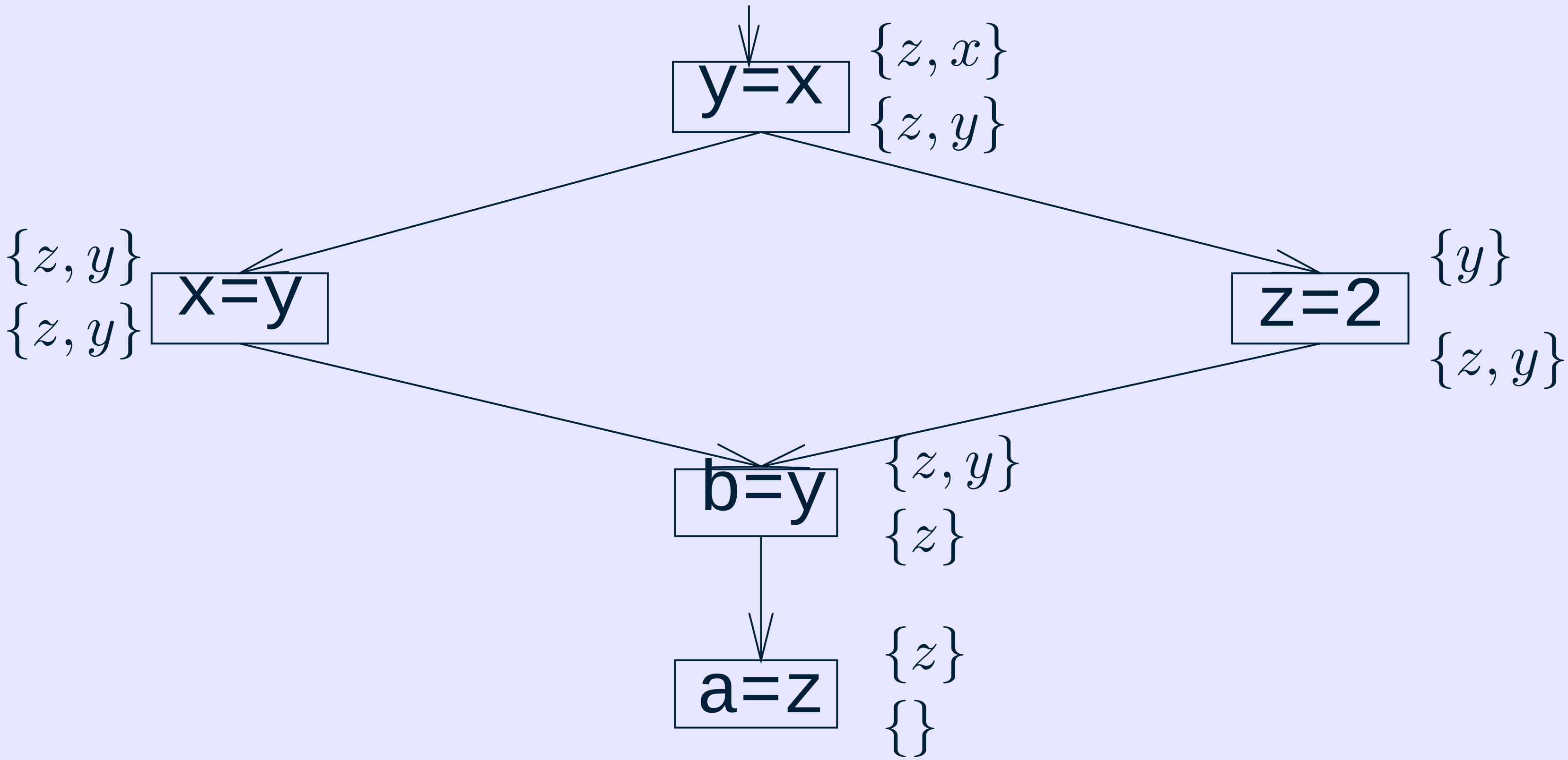
Outline: Soot Flow Analysis Examples

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

- live locals
- branched nullness testing

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<Unit,Set>
```

soot.toolkits.scalar.BackwardFlowAnalysis

Step 2: Abstraction domain

Domain for Live Variables: sets of `Locals`

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

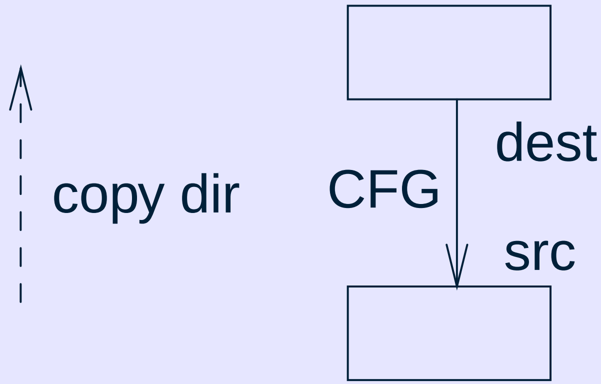
- Partial order is subset inclusion
- Merge operator is union

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

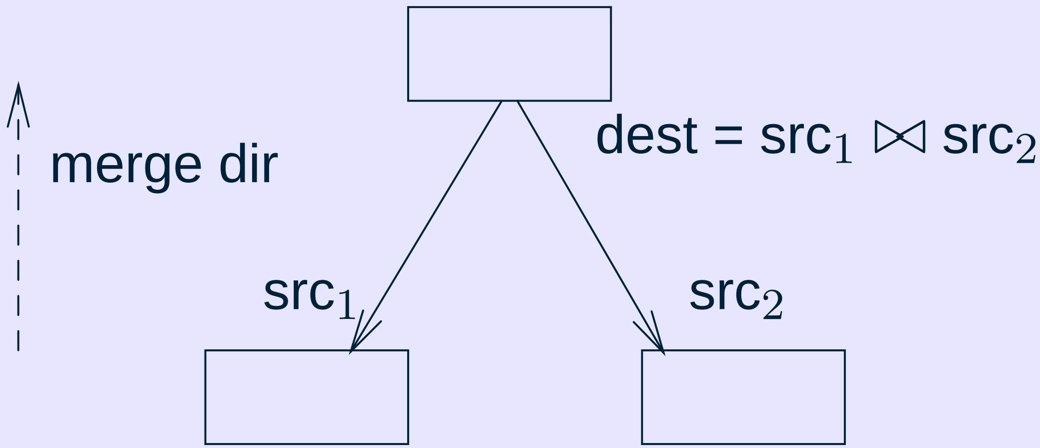
soot.toolkits.scalar.ArraySparseSet

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(Set src1, Set src2,
           Set dest);
 void copy(Set src, Set dest);
```

Flow Sets and Soot

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

Impls: `ToppedSet`, `ArraySparseSet`, `ArrayPackedSet`

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

```
// c = a ∪ b
a.union(b,c);
```

```
// d = c̄
c.complement(d);
```

```
// d = d ∪ {v}
d.add(v);
```

soot.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`)

soot.toolkits.scalar.*Set

Step 2: `copy()` for live variables

```
protected void copy(Set src,
                    Set dest) {
  dest.clear();   dest.addAll(src);
}
```

Step 2: `merge()` for live variables

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

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

Often, you may want to implement a more exotic merge.

Step 3: Flow equations

Goal: At a unit like `x = y * z`:

kill def x;

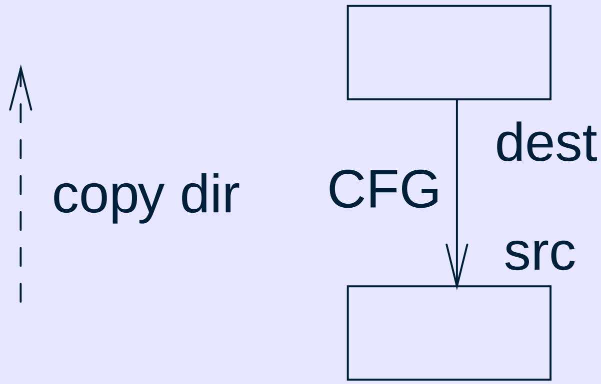
gen uses y, z.

How? Implement this method:

```
protected void flowThrough
                         (Set srcValue,
                          Unit u,
                          Set destValue)
```

Step 3: Copying

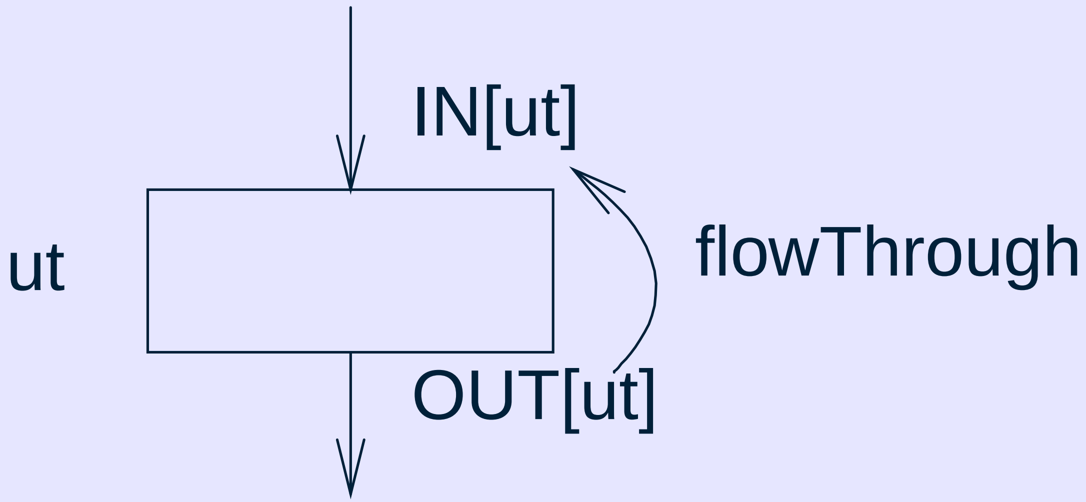
Need to copy `src` to `dest` to allow manipulation.




```
dest.clear();
dest.addAll(src);
```

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.

Step 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 );
}
```

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
    ```
    Set newInitialFlow()
        {return new HashSet();}
    ```
  - Create initial sets for exit nodes
    ```
    Set entryInitialFlow()
        {return new HashSet();}
    ```

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

Example 2: VeryB

```
class VeryBusyExpressionAnalysis
      extends BackwardFlowAnalysis {
  [...]
}
```

VeryB - Constructor

```
public VeryBusyExpressionAnalysis(
        DirectedGraph g) {
    super(g);
    doAnalysis();
}
```

VeryB - Merge

```
protected void merge(Object in1,
                     Object in2,
                     Object out) {
  FlowSet inSet1 = (FlowSet)in1,
          inSet2 = (FlowSet)in2,
          outSet = (FlowSet)out;
  inSet1.intersection(inSet2,
                      outSet);
}
```

VeryB - Copy

```
protected void copy(Object source,
                    Object dest) {
  FlowSet srcSet = (FlowSet)source,
          destSet = (FlowSet)dest;
  srcSet.copy(destSet);
}
```

VeryB - Flow

```
protected void flowThrough(Object in,
                           Object node,
                           Object out) {
  FlowSet inSet = (FlowSet)source,
         outSet = (FlowSet)dest;
  Unit u = (Unit)node;

  kill(inSet, u, outSet);
  gen(outSet, u);
}
```

VeryB - Gen

```
private void gen(FlowSet outSet,
                 Unit u) {
  for (ValueBox useBox: u.getUseBoxes()) {
    if (useBox.getValue()
          instanceof BinopExpr)
      outSet.add(useBox.getValue());
    }
  }
}
```

VeryB - Kill

```
private
void kill(FlowSet in, Unit u, FlowSet out) {
  FlowSet kills = (FlowSet)emptySet.clone();
  for (ValueBox defBox: u.getUseBoxes()) {
    if (defBox.getValue() instanceof Local) {
      for (BinopExpr e: in) {
        for (ValueBox useBox: e.getUseBoxes()) {
          if (useBox.getValue() instanceof Local
              && useBox.getValue().equivTo(
                  defBox.getValue()))
            kills.add(e);
        }}}}
  in.difference(kills, out);
}
```

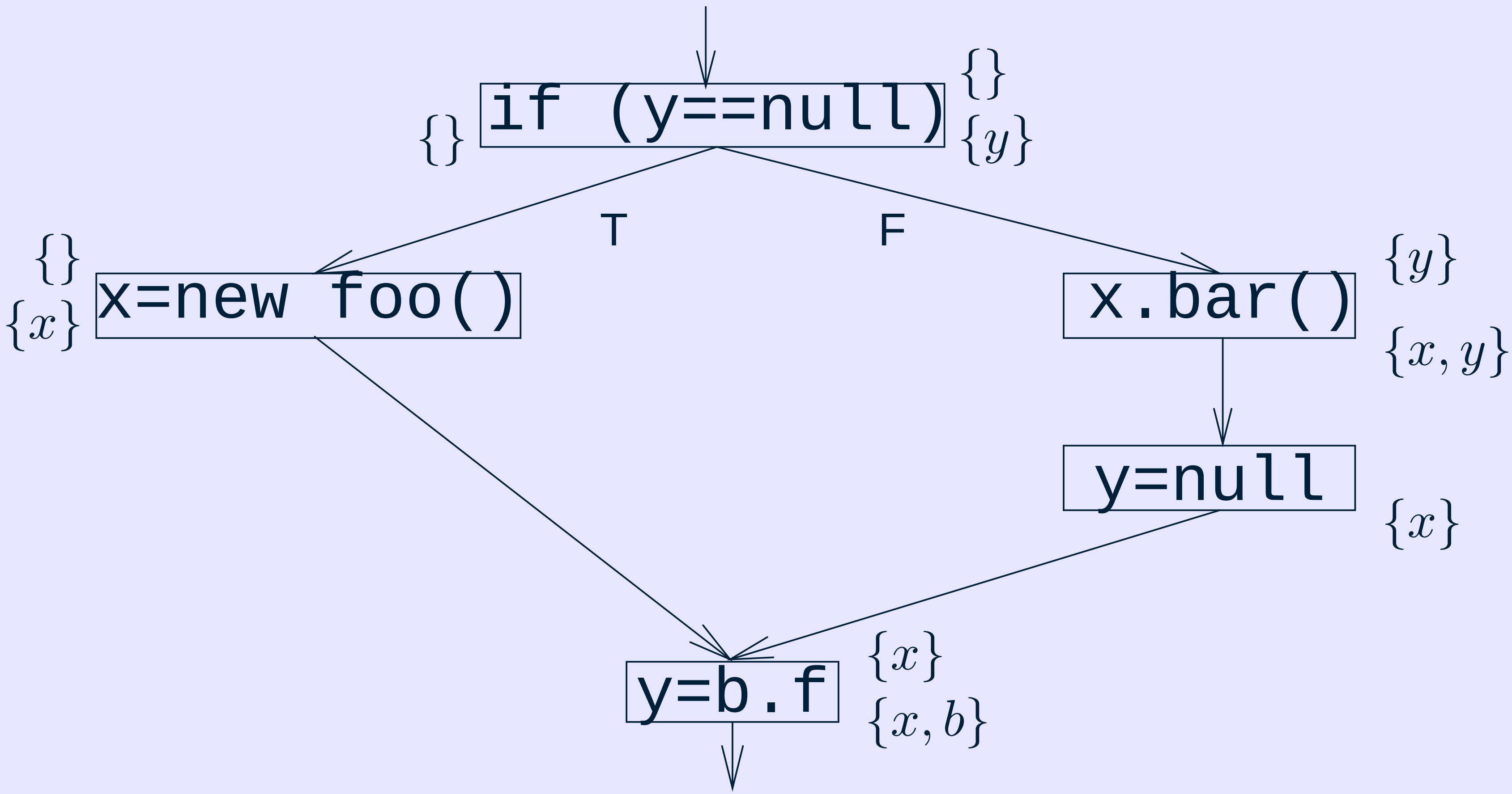
VeryB - Initial State

```
protected Object entryInitialFlow() {
  return new ValueArraySparseSet();
}

protected Object newInitialFlow() {
  return new ValueArraySparseSet();
}
```

Example 3: Branched Nullness

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



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<Unit,FlowSet> {
```

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

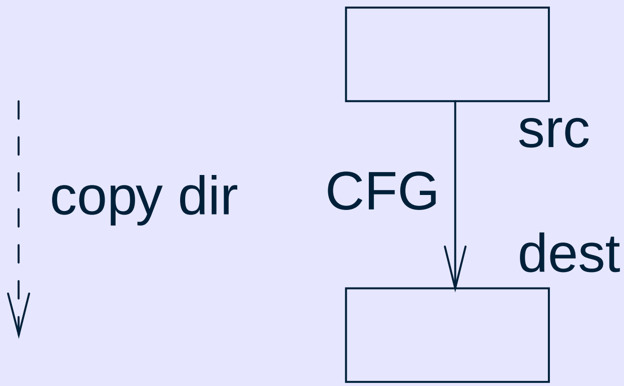
This time, use `ArraySparseSet` to implement:

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

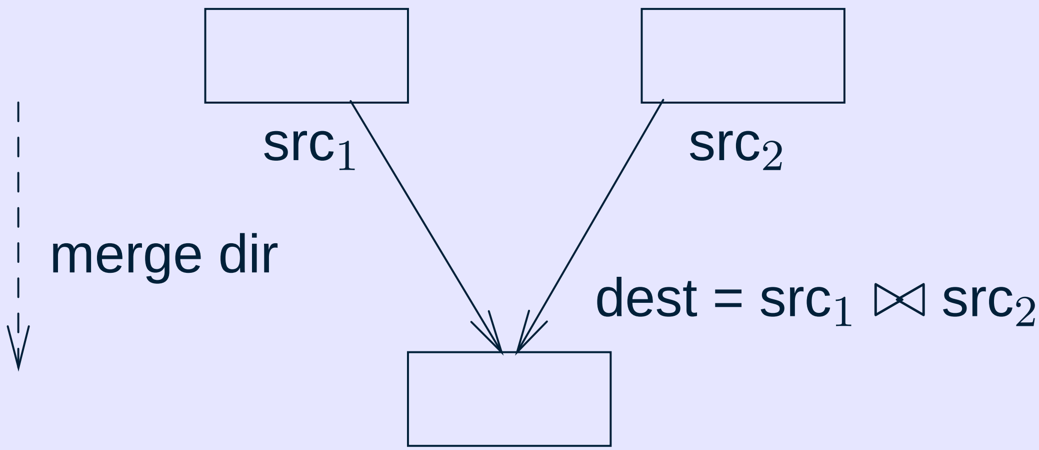
* see soot.jimple.toolkits.annotation.nullcheck.BranchedRefVarsAnalysis

Implementing an Abstraction

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



`copy()` 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(FlowSet src,
                    FlowSet dest) {
  src.copy(dest);
}
```

Use `copy()` method from `FlowSet`.

Step 2: `merge()` for nullness

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

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

```
void merge(...) {
    srcSet1.intersection(srcSet2,
                         destSet);
}
```

Step 3: Branched Flow Function

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

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

`fallOut` is a one-element list.

`branchOuts` contains a `FlowSet` for each non-fallthrough successor.

Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.

Step 3: Flow equations

We do the following things in our flow function:

- Create copy of src set.
- Remove kill set (defined `Locals`).
  `y` 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.)
  ```
  FlowSet newInitialFlow() {
      { return fullSet.clone(); }
  ```
- Create entry sets (`emptySet` from constr.)
  ```
  FlowSet 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`

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`:

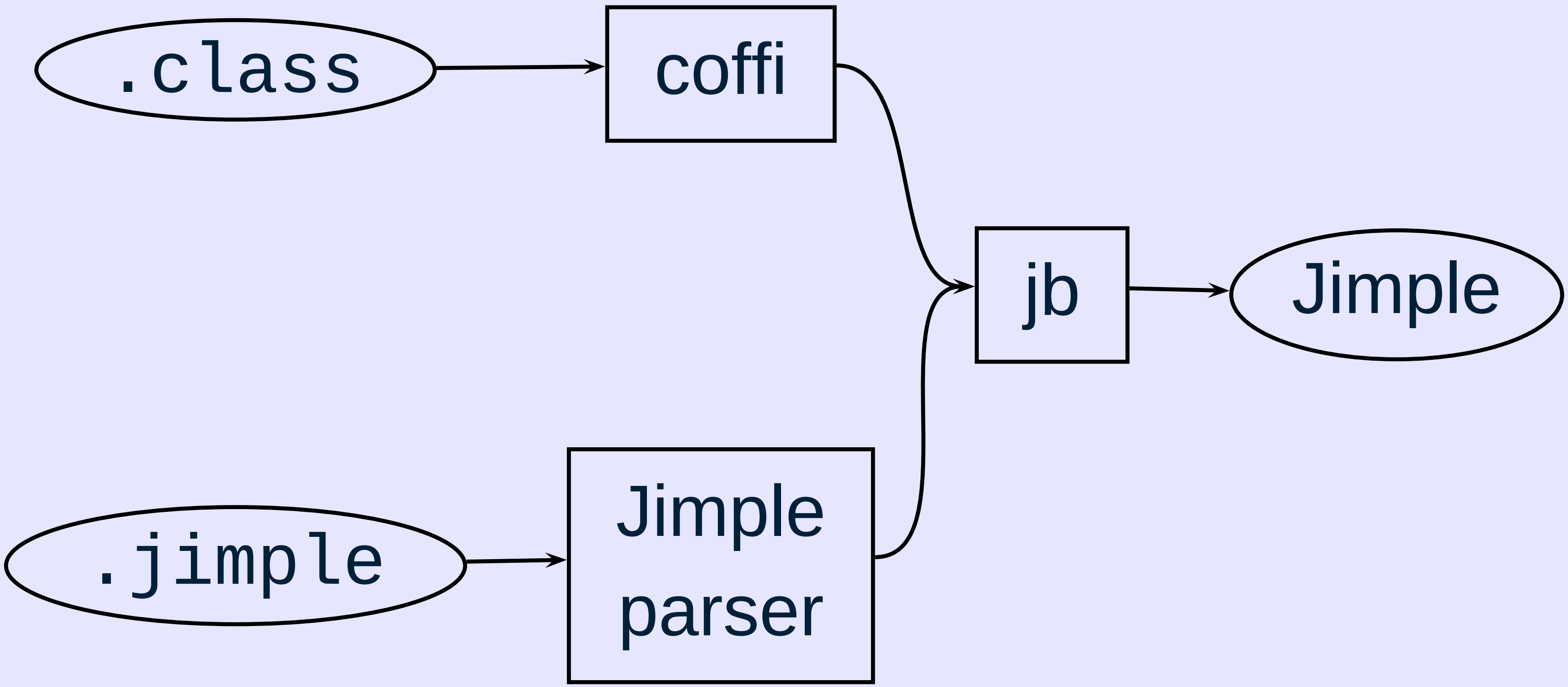
```
Pack jtp = G.v().PackManager().
                   getPack("jtp");
jtp.add(new Transform("jtp.nt",
          new NullTransformer()));
jtp.add(new Transform("jtp.nac",
          new NullnessAnalysisColorer()));
```

soot.Pack

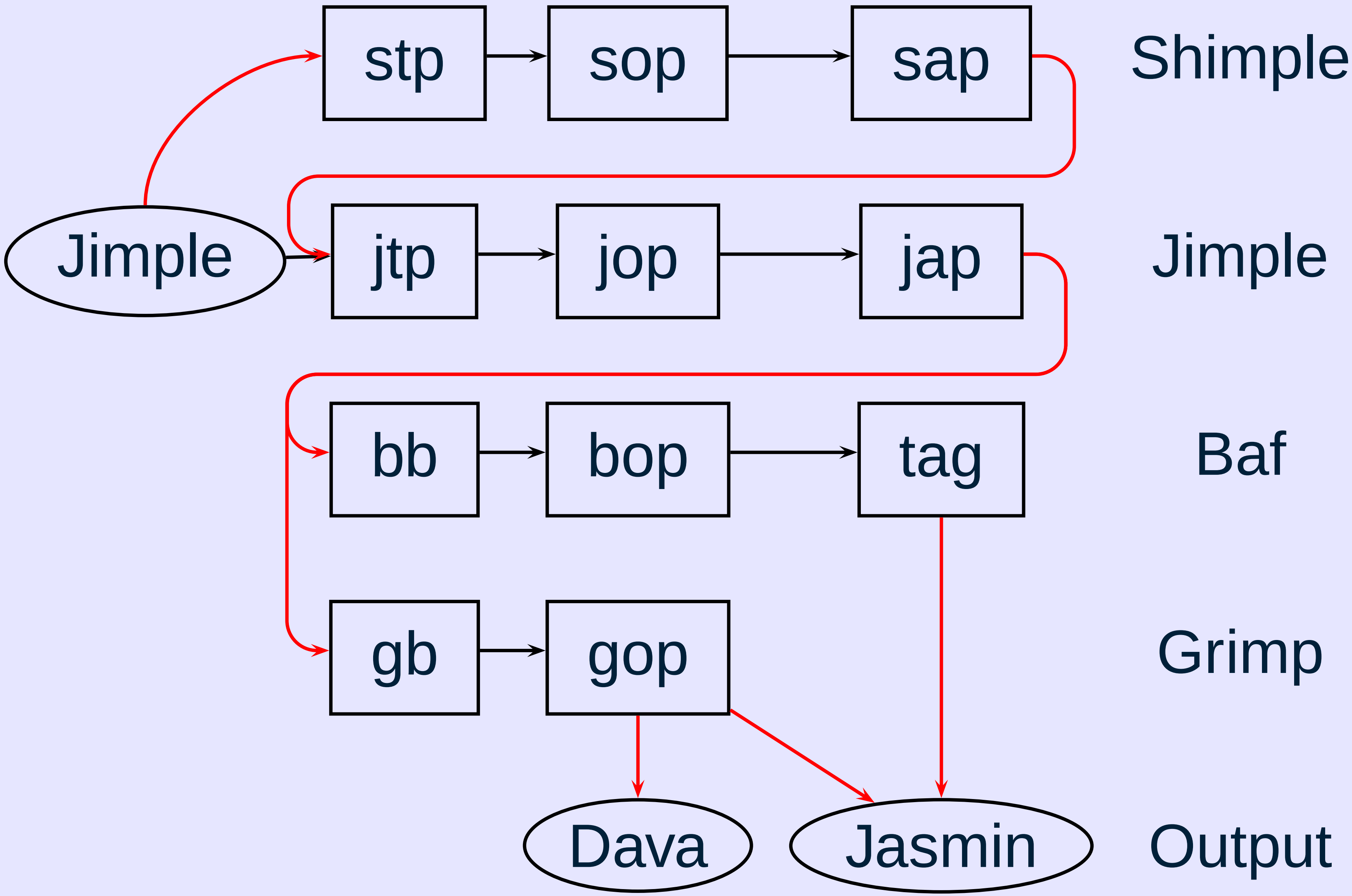
Running Soot more than once

- 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

The 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

Conclusion

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

Resources

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

Theses and papers:

`www.sable.mcgill.ca/publications/`

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

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

Mailing lists:

`www.sable.mcgill.ca/soot/#mailingLists`

Soot in a Course:

`www.sable.mcgill.ca/~hendren/621/`