Tools and Analyses for Optimizing Java Bytecode

Laurie Hendren

Raja Valleé-Rai

Etienne Gagnon

Vijay Sundaresan

Chrislain Razafimahefa

Patrick Lam

Sable[a] Research Group

McGill University

Montreal, Canada

`www.sable.mcgill.ca`

[a]Sable Island is a mile-wide crescent of sand known as the "graveyard of the Atlantic".

Outline:

- Motivation and Overview
- Introduction to Java Bytecode
- Soot Framework and the Jimple Intermediate Rep.
- Example 1: live variable analysis
- Example 2: virtual method resolution
- Future Work

Motivation:

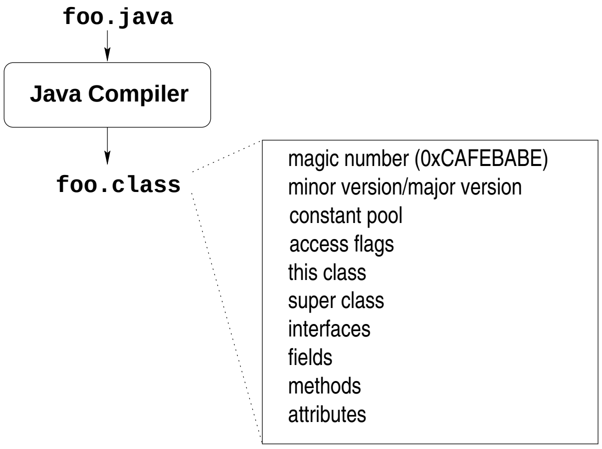
- Sun claims that Java is:
  - simple
  - object-oriented
  - distributed
  - interpreted
  - robust
  - secure
  - architecture neutral
  - portable
  - multithreaded
  - dynamic

  ... and also high-performance?
- In fact, considerable work is required to make Java a high-performance language.

Sable group's goals are to:

- develop tools for analyzing and transforming Java bytecode;
- develop static analyses for automatic optimization of Java bytecode;
- develop runtime optimizations that can take advantage of static information; and
- in general, have fun.

Compiling Java to Class Files



Compiling and Decompiling Class Files

Avg.java

```
class Avg {

float average(float x, float y, float z)
  { return( (x+y+z)/3.0f ); }

}
```

javac Avg.java

Avg.class

```
...
Method float average(float,float,float)
  0 fload_1
  1 fload_2
  2 fadd
  3 fload_3
  4 fadd
  5 ldc #1 <Real 3.000000>
  7 fdiv
  8 freturn
```

mocha.Decompile Avg.class

Avg.mocha

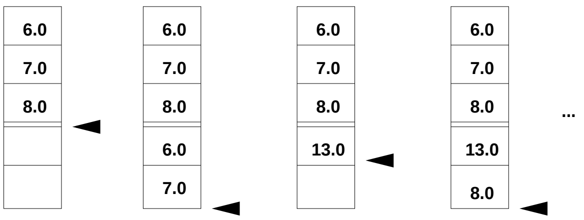
```
class Avg {

float average(float f1, float f2, float f3)
  { return (f1 + f2 + f3) / 3; }

}
```


Java Virtual Machine

Avg.class

```
...
Method float average(float,float,float)
  0 fload_1
  1 fload_2
  2 fadd
  3 fload_3
  4 fadd
  5 ldc #1 <Real 3.000000>
  7 fdiv
  8 freturn
```

Suppose we invoke average with: average(6.0,7.0,8.0)



Bytecode Verification

- Bytecode cannot be trusted to be well-formed and well-behaved.
- Before executing any bytecode that is received over the network, it should be verified.
- Verification is performed partly at class loading time, and partly at run-time.
- At load time, dataflow analysis is used to approximate the number and type of values in locals and on the stack.

Interesting Properties of Verified Bytecode

- Each instruction must be executed with the correct number and types of arguments on the stack, and in locals (on all execution paths).
- At any program point p, the stack is the same size along all execution paths.
- No local variable can be accessed before it has been assigned a value.

Interpreting Java

- When a method is invoked, a `classloader` must find the correct class, and check that it contains an appropriate method. This may require traversing up the inheritance hierarchy. If the class is not found, then an exception is raised.
- If the method has not yet been loaded, then it may be verified (remote classes). If it fails verification, an exception is raised.
- After loading and verification, either:
  - parameters are placed in the appropriate locals, and interpretation of the method body begins; or
  - the bytecode for the method is translated to native code (only for the first invocation), the appropriate parameters are loaded and control is given to the native code.

Execution times for JIT vs Interpreter - Sun JDK (win95)

	1.1.7 JIT		1.1.7 no JIT	
	(micro.)	LVA	(micro.)	LVA
Local (i = n)	0.0069	1.00	0.0893	1.00
Static (f = n)	0.0524	7.48	0.1022	1.15
Instance(this.f = n)	0.0518	7.51	0.1347	1.51
Array (a[i] = n)	0.0520	7.54	0.1916	2.51
New (new Object())	3.7066	531.43	3.36	37.72
Array (new int[1000])	117.0349	17005.62	117.72	1318.89
Static (A.m(i))	0.0064	1.28		
Virtual (a.m(i))	0.0296	4.38	1.24	13.97
Interf. (a.n(i))	0.0438	6.40	0.86	9.72
Sync. (s.m(i))	1.9155	278.70	2.74	30.78
Cast (a = (D) b)	0.0067	0.98	0.37	4.16
InstanceOf	0.0034	0.49	0.37	4.16
Try (try.. catch..)	0.0269	3.90	0.25	2.78
Caught (try.. catch..)	52.5157	7648.09	100.48	1125.72

Analysis and transformation of 3-addr code

```
f()
{ int x,y,z;

  x = 1;            {(x,1)}
  y = 2;            {(x,1),(y,2)}
  z = x + y;        {(x,1),(y,2),(z,3)}
  return(z);
}
```

```
f()
{ int x,y,z;

  x = 1;
  y = 2;
  z = 3;
  return(3);
}
```

```
f()
{ int x,y,z;



  return(3);
}
```

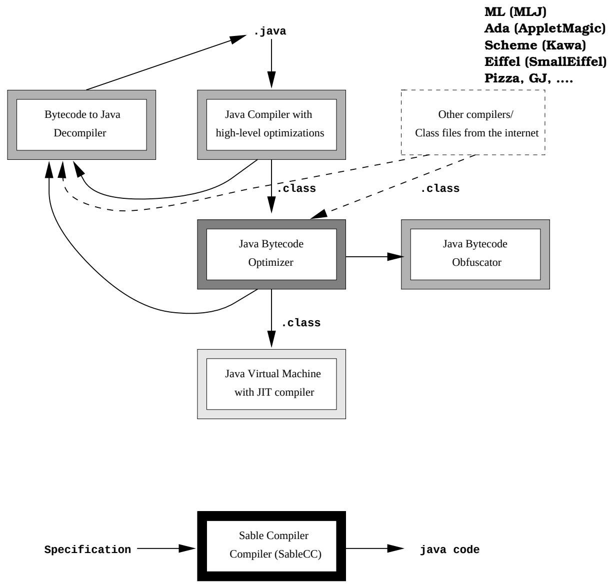
Analysis and transformation of stack code

```
iconst_1   {(s0,1)}
istore_0   {(l0,1)}
iconst_2   {(l0,1),(s0,2)}
istore_1   {(l0,l),(l1,2)}
iload_0    {(l0,1),(l1,2),(s0,1)}
iload_1    {(l0,1),(l1,2),(s0,1),(s1,2)}
iadd       {(l0,1),(l1,2),(s0,3)}
istore_2   {(l0,1),(l1,2),(l3,3)}
iload_2    {(l0,1),(l1,2),(l3,3),(s0,3)}
ireturn
```

```
iconst_1
istore_0
iconst_2
istore_1
iconst_1
iconst_2
pop
pop
iconst_3
istore_2
iconst_3
ireturn
```

```
iconst_3
ireturn
```


Overview of the Sable System:



Soot Framework:

- A publically-available framework for working with Java bytecode.
- Currently consists of three intermediate representations:

  Baf: a representation of bytecode; and

  Jimple: a three-address representation like bytecode, but with no stack

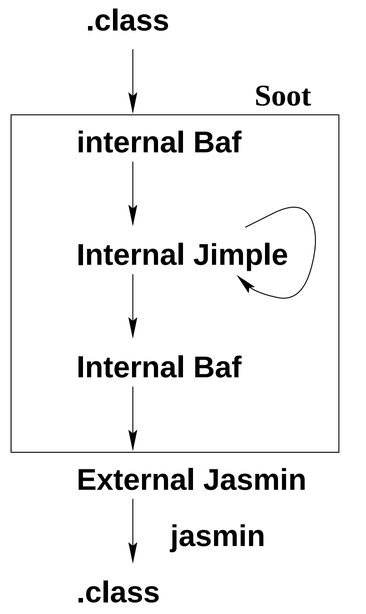
  Grimp: Jimple, but with aggregated expressions
- Each representation will eventually have both an internal and external form.
- Currently **Baf** is only used within the Soot framework.
- Several projects have already used **Jimple**.
- **Grimp** is mostly used for decompilation

Jimple is a three-address code:

- Each statement is **simple**;
- Statements refer only to **typed**, named variables;
- An example of bytecode and generated Jimple:

```
iload 0                 int x, y, z;
iload 1
iadd         -->        z = x + y;
istore 2
```

Bytecode → Jimple → bytecode:



Simple example of using Soot:

`java PrintClass MyFile`

```
import ca.mcgill.sable.soot.*;
import ca.mcgill.sable.soot.jimple.*;
import java.io.*;

// Prints the given class in .jimple format.
public class PrintClass
{ public static void main(String[] args)
  { // Retrieve and print class.
    SootClassManager cm = new SootClassManager();
    SootClass sClass = cm.getClass(args[0]);
    PrintWriter out =
                  new PrintWriter(System.out, true);

    sClass.printTo(new BuildBody(Jimple.v(),
                     new StoredBody(ClassFile.v())),
                   out);
  }
}
```


Adding Profiling Counters:

```
...                       tmpLocal = gotoCounter;
                          gotoCounter = tmpLocal +1;
goto L1;                  goto L1;
```

```
StmtList stmtList = body.getStmtList();
ListIterator stmtIt = stmtList.listIterator();
while(stmtIt.hasNext())
{ Stmt s = (Stmt) stmtIt.next();
  if (s instanceof GotoStmt)
  { // Back up before the GotoStmt
    stmtIt.previous();
    // insert "tmpLocal = gotoCounter;"
    stmtIt.add(Jimple.v().newAssignStmt(tmpLocal,
       Jimple.v().newStaticFieldRef(gotoCounter)));
    // skip over statement just added
    stmtIt.next();
    // insert "gotoCounter = tmpLocal + 1L;"
    stmtIt.add(Jimple.v().newAssignStmt(
      Jimple.v().newStaticFieldRef(gotoCounter),
      Jimple.v().newAddExpr(tmpLocal,
                       LongConstant.v(1L))));
    // skip over statement just added
    stmtIt.next();
    // skip over the goto
    stmtIt.next();
  }
}
```


Steps in producing Jimple:

Step 1: Produce verbose, typeless Jimple.

```
     .numlocals 2        unknown a, b;
     .maxdepth 2         unknown op0, op1;

S1:  iload 0             op0 = a;
S2:  iload 0             op1 = a;
S3:  imul                op0 = op0 * op1;
S4:  istore 1            b = op0;
```

Step 2: Compact using const. and copy prop, and dead code elimination.

```
  Before                After

op0 = a
op1 = b               x = a + b
op0 = a + b
x = op0

op0 = a               call f(a, b)
op1 = b
call f(op0, op1)
```


Cannot always eliminate named stack locations:

```
       /\
iload 0     iload 2
iload 1     iload 3
       \/
       /\
iadd         imul
istore 4     istore 4
       \/
```

... becomes ....

```
           /\
op0 = a       op0 = c
op1 = b       op1 = d
           \/
           /\
op0=op0+op1   op0=op0*op1
e=op0         e=op0
           \/
```


Step 3: Splitting the locals based an U/D and D/U webs.

```
op0 = "hello world";
string = op0;
op0 = 1;
unit = op0;
```

... becomes ...

```
op0 = "hello world";
string = op0;
op1 = 1;
unit = op1;
```


Step 4: Typing the local variables.

- This step turned out to be non-trivial, we will come back to it.

Step 5: Packing the variables.

- Use a register-allocation style algorithm to assign all variables of the same type to the fewest number of names.

```
              splitting
x = y + z;    -------->    x1 = y + z;
print(x);                  print(x1);
x = a + b;                 x2 = a + b;
print(x);     <-------     print(x2);
               packing
```

```
public int stepPoly(int x)
{ if (x < 0)
    { System.out.println("error"); return -1; }
  else if (x <= 5)
    return x * x;
  else
    return x * 5 + 16;
}

public int stepPoly(int)
{ java.io.PrintStream r1;
  Example r0;
  int i0, i1, i2, i3;
  r0 := @this;
  i0 := @parameter0;
  if i0 >= 0 goto label0;
  r1 = java.lang.System.out;
  r1.println("error");
  return -1;
label0:
  if i0 > 5 goto label1;
  i1 = i0 * i0;
  return i1;
label1:
  i3 = i0 * 5;
  i2 = i3 + 16;
  return i2;
}
```


Typing Jimple:

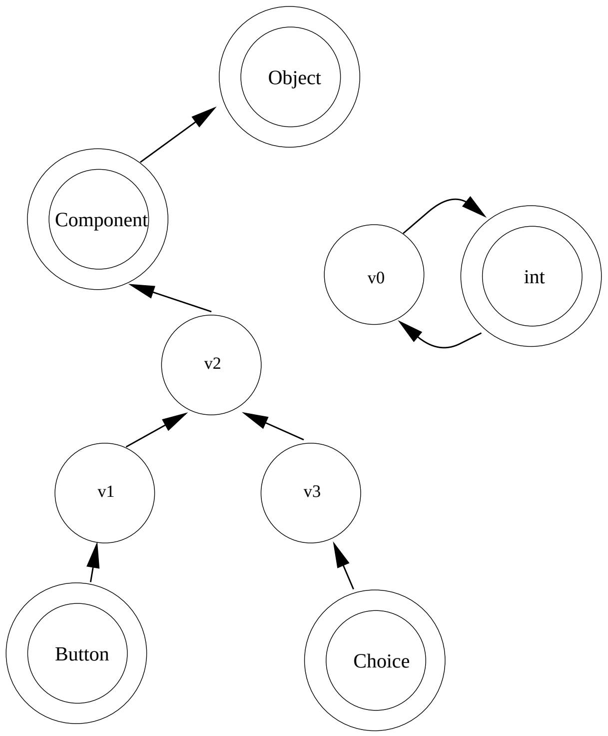
- Need to find a correct type assignment for each local variable declaration in Jimple.
- We can type one method at a time.
- Some type constraints come from signatures in the bytecode.
- Other constraints are due to assignments.
- In general, this type assignment problem is NP-complete.
- In practice, a cheap polynomial solution works for all methods analyzed so far (> 16,000 methods).

- **Type constraints expressed as a graph:**
  - *soft* nodes are Jimple variables, type needs to be defined
  - *hard* nodes are types in the type hierarchy
  - directed edges represent subclass relationships imposed by Jimple statements
- Type assignment proceeds by collapsing nodes.
- When a *hard* node is collapsed with a *soft* node, the type assignment is made.

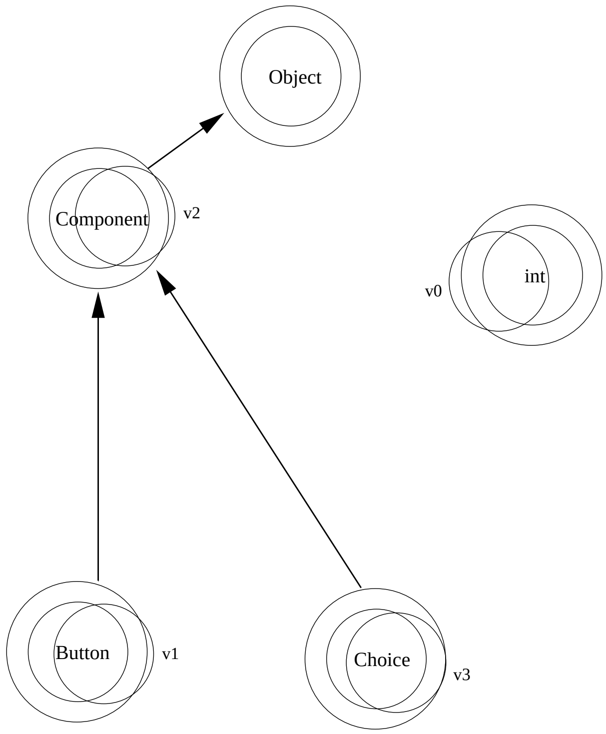
Example untyped Jimple:

```
import java.awt.*;

class Test
{ static Component method(boolean b)
  { Component c = new Button();
    if(b) c = new Choice();
    Object o = c;
    o.notify();
    return c;
  }
}

class Test extends java.lang.Object
{ static java.awt.Component method(boolean param0)
  { unknown_type v0;
    unknown_type v1;
    unknown_type v2;
    unknown_type v3;
    v0 := param0;
    v1 = new java.awt.Button;
    specialinvoke v1.[void java.awt.Button.<init>()]();
    v2 = v1;
    if v0 == 0 goto label0;
    v3 = new java.awt.Choice;
    specialinvoke v3.[void java.awt.Choice.<init>()]();
    v2 = v3;
 label0:
    virtualinvoke v2.[void java.lang.Object.notify()]();
    return v2;
  }
}
```

Example Graph, transitive edges removed:



Solution:



Example Typed Jimple:

```
import java.awt.*;

class Test
{ static Component method(boolean b)
  { Component c = new Button();
    if(b) c = new Choice();
    Object o = c;
    o.notify();
    return c;
  }
}

class Test extends java.lang.Object
{ static java.awt.Component method(boolean param0)
  { int v0;
    java.awt.Button v1;
    java.awt.Component v2;
    java.awt.Choice v3;
    v0 := param0;
    v1 = new java.awt.Button;
    specialinvoke v1.[void java.awt.Button.<init>()]();
    v2 = v1;
    if v0 == 0 goto label0;
    v3 = new java.awt.Choice;
    specialinvoke v3.[void java.awt.Choice.<init>()]();
    v2 = v3;
 label0:
    virtualinvoke v2.[void java.lang.Object.notify()]();
    return v2;
  }
}
```


Real methods are very simple to solve:

	methods (#)
Cyclic Relation Elimination	6206
Single Relation Elimination	10747
Exponential Algorithm	0
TOTAL	16953

Jimple provides:

- a typed 3-address representation
- methods to access and modify the representation
- API's to support flow analysis

  Example 1: live variable analysis

  Example 2: virtual method resolution

The intraprocedural flow analysis API

```
public abstract class FlowAnalysis
     extends java.lang.Object
{ // Fields 2
  protected Map stmtToAfterFlow;
  protected Map stmtToBeforeFlow;

  // Constructors 1
  public FlowAnalysis(StmtGraph);

  // Methods 8
  protected abstract void copy(Object, Object);
  protected abstract void doAnalysis();
  protected abstract void flowThrough(Object, Stmt, Object);
  public Object getFlowAfterStmt(Stmt);
  public Object getFlowBeforeStmt(Stmt);
  protected abstract boolean isForward();
  protected abstract void merge(Object, Object, Object);
  protected abstract Object newInitialFlow();
}

public abstract class BackwardFlowAnalysis
     extends ca.mcgill.sable.soot.jimple.FlowAnalysis
{ // Constructors 1
  public BackwardFlowAnalysis(StmtGraph);

   // Methods 2
   protected void doAnalysis();
   protected boolean isForward();
}
```


Live Variable Analysis:

```
class SparseLiveLocalsAnalysis extends BackwardFlowAnalysis
{ FlowSet emptySet;
  Map stmtToGenerateSet;
  Map stmtToKillSet;

  SparseLiveLocalsAnalysis(StmtGraph g)
    { super(g);
      emptySet = new ArraySparseSet();
      // Create Kill sets.
      { stmtToKillSet = new HashMap(g.size()*2+1, 0.7f);
        Iterator stmtIt = g.iterator();
        while(stmtIt.hasNext())
          { Stmt s = (Stmt) stmtIt.next();
            FlowSet killSet = (FlowSet) emptySet.clone();
            Iterator boxIt = s.getDefBoxes().iterator();
            while(boxIt.hasNext())
              { ValueBox box = (ValueBox) boxIt.next();
                 if(box.getValue() instanceof Local)
                   killSet.add(box.getValue(), killSet);
               }
             stmtToKillSet.put(s, killSet);
          }
       }
```


(cont'd)

```
     // Create generate sets
     { stmtToGenerateSet = new HashMap(g.size()*2+1,0.7f);
       Iterator stmtIt = g.iterator();
       while(stmtIt.hasNext())
         { Stmt s = (Stmt) stmtIt.next();
           FlowSet genSet = (FlowSet) emptySet.clone();
           Iterator boxIt = s.getUseBoxes().iterator();
           while(boxIt.hasNext())
             { ValueBox box = (ValueBox) boxIt.next();
               if(box.getValue() instanceof Local)
                genSet.add(box.getValue(), genSet);
              }
           stmtToGenerateSet.put(s, genSet);
         }
      }
    doAnalysis();
  }

  protected Object newInitialFlow()
  { return emptySet.clone(); }

  protected void flowThrough(Object inValue,
                                Stmt stmt, Object outValue)
  { FlowSet in = (FlowSet) inValue, out = (FlowSet) outValue;
    // Perform kill
    in.difference((FlowSet) stmtToKillSet.get(stmt), out);
    // Perform generation
    out.union((FlowSet) stmtToGenerateSet.get(stmt), out);
  }
}
```

Example 2: Virtual Method Resolution

- Analyze a complete application (all class files must be statically available).
- Estimate, for each invokevirtual and invokeinterface, the set of possible methods that could be called.
- Better virtual method resolution leads to:
  - a more accurate call graph
  - a smaller code base (eliminate unused methods)
  - better subsequent analysis
  - improved runtime performance

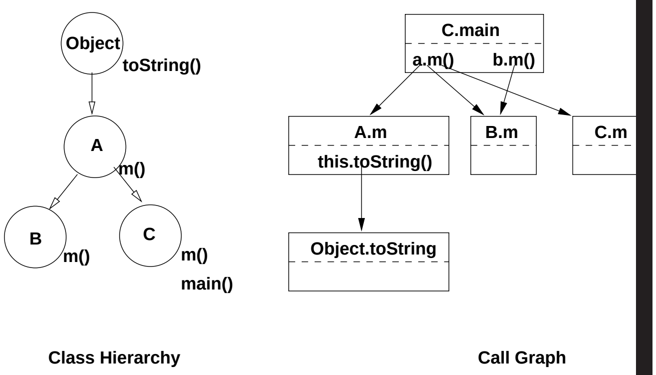
Possible Solutions:

- hierarchy analysis [Dean, Grove and Chambers, Fernandez];
- rapid type analysis [Bacon and Sweeney] ; and
- reaching type analysis (new).

Class Hierarchy Analysis and the Conservative Call Graph:

```
class A extends Object {
  String m() {
    return(this.toString());
  }
}

class B extends A {
  String m() { ... }
}

class C extends A {
  String m() { ... }
  public static void main(...) {
   A a = new A();
   B b = new B();
   String s;
   ...
   s = a.m();
   s = b.m();
  }
}
```


Hierarchy, Conservative Call Graph, Rapid Type Analysis



- *Rapid type analysis* refines hierarchy analysis by excluding calls that are due to classes that have no instances.

Collecting all instantiated classes for rapid type analysis.

```
{ List newExprs = new ArrayList();
  ...
  // iterate over call graph nodes
  while ( iter.hasNext() )
  { MethodNode tempMN = (MethodNode) iter.next();
    SootMethod currmethod = tempMN.getMethod();
    JimpleBody listBody = ( JimpleBody )
                          currmethod.getBody ( Jimple.v() );
    Iterator stmtIter = listBody.getStmtList().iterator();
    // iterate over stmts of a method
    while ( stmtIter.hasNext() )
      { Stmt stmt = (Stmt) stmtIter.next();
        stmt.apply(
          new AbstractStmtSwitch() {
            public void caseAssignStmt(AssignStmt s){
              if ((s.getRightOp() instanceof NewExpr) ||
                  (s.getRightOp() instanceof NewArrayExpr) ||
                  (s.getRightOp() instanceof NewMultiArrayExpr) )
                newExprs.add( s.getRightOp() );
            }});
     }
  }
}
```


Reaching type analysis:

- estimates the set of allocations that can reach the receiver of an `invokevirtual` or `invokeinterface`;
- will be used on large applications; and thus
- must be efficient and scale linearly (flow-insensitive).

Build a *type propagation graph*, where:

- nodes are created for:
  - each field;
  - each local;
  - each parameter (including the implicit first parameter `this`); and
  - one node representing the return value for each method.
- edges are created for assignments $lhs \leftarrow rhs$ (including assignments due to parameter passing and returning values).
- edges for calls are based on calls in the conservative call graph

Types are propagated by:

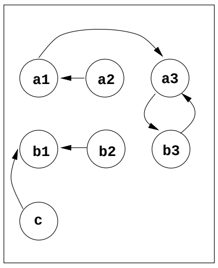
1. Initialize nodes due to `new` statements.
2. Collapse strongly-connected components.
3. Propagate types on resulting DAG.

Two variations, *variable-type analysis* and *declared-type analysis*.

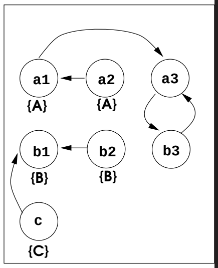
Variable-type analysis

```
A a1, a2, a3;
B b1, b2, b3;
C c;

a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```

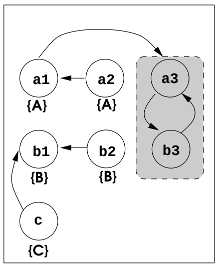
(a) Program



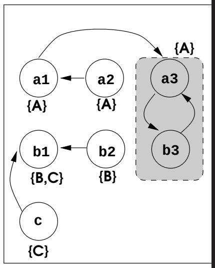
(b) Nodes and Edges



(c) Initial Types



(d) Strongly-connected components

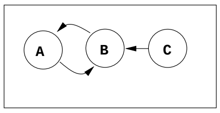


(e) final solution

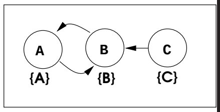
Declared-type analysis

```
A a1, a2, a3;
B b1, b2, b3;
C c;

a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```

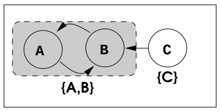
(a) Program



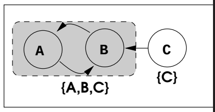
(b) Nodes and Edges



(c) Initial Types



(d) Strongly-connected components



(e) final solution

Benchmark Characteristics

Source	Benchmark	# Jimp (whole)	# Jimp (bench)	Hier (whole)	Hier (bench)
java	sablecc	68575	24621	3.2	2.3
java	soot	63506	33396	3.3	2.1
java	raytrace	49239	5347	3.0	1.3
java	jess	56163	11137	2.8	1.3
java	compress	46619	2727	3.0	1.1
java	jack	55107	11215	3.0	1.6
java	javac	69585	25304	3.5	3.2
eiffel	illness	29568	1372	3.2	1.8
ml	nucleic	33096	4900	3.1	1.6
ml	lexgen	33397	5201	3.1	1.4
ml	ray	34186	3721	3.1	1.6
ada	rudstone	75250	31413	2.9	1.2
pizza	pizza comp.	73130	42805	3.0	1.7

Static improvement over the conservative call graph for whole application:

	Rapid Type (RTA)	Variable Type (VTA)
Call Edges Removed	5% - 22%	12% - 35%
Calls Resolved	3% - 23%	19% - 54%

Dynamic Virtual Calls resolved to 1 target, Benchmark Only:

Source	Benchmark	CHA	RTA	VTA	Profile
java	sablecc	86%	86.5%	89.7%	95.5%
java	soot	29%	29%	41%	66.6%
java	jack	86.3%	87%	98.5%	98.5%
java	javac	65.9%	65.9%	72.9%	90.1%
ml	nucleic	0%	0%	99.1%	99.1%
ml	lexgen	81.8%	81.8%	100%	100%
pizza	pizza comp.	75.5%	75.5%	89%	94%

Size of Data Structures

- size scales linearly with number of Jimple statements.
- graphs are relatively sparse (2-5 as many edges as nodes).
- SCC does not reduce the size of the graph significantly.
- variable-type analysis had 2-4 times as many nodes as declared-type analysis.

Summary

- Soot is a framework for analyzing and optimizing bytecode.
- Jimple is a three-address representation.
- Converting from bytecode to Jimple requires a typing algorithm.
- Soot/Jimple can be used for implementing a variety of analyses.
- check out `www.sable.mcgill.ca`