Points-to Set Analysis

E0 227: Program Analysis and Verification 2025

Alan Jojo, Atharv Desai

Programming Languages Lab
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Project

- Given a Java program, implement a tool that performs *Points-to analysis*.
- *Points-to analysis* is a static program analysis, which reports the potential allocations sites associated with a variable at a program point at run-time.
- Project is in two phases.

Phase I

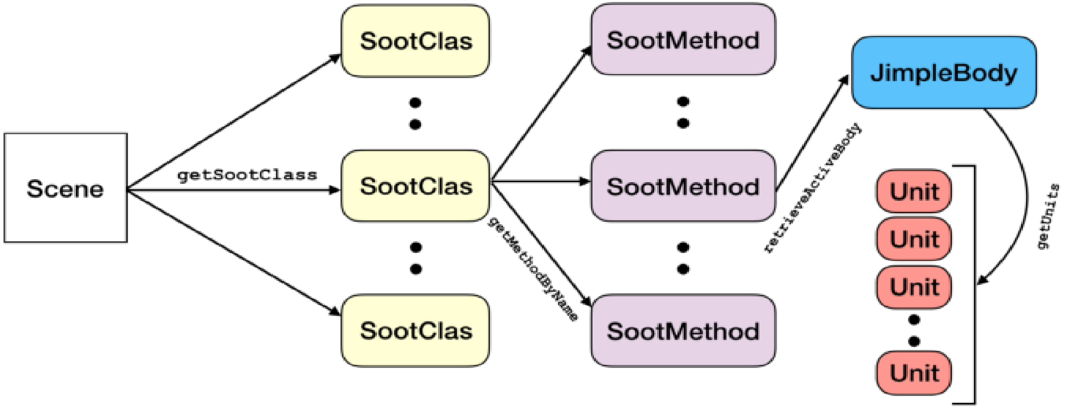
- Problem Statement: Intra-procedural *Points-to Analysis* (using Kildall's fixed-point algorithm)
- Input : A compiled set of *.class* files from a *.java* file.
- Assumptions :
    - Method calls returning reference to objects to be considered as allocation sites.
    - Ignore rest of Method Calls.
    - Ignore Exceptional control flow edges.
    - Ignore *static* class fields.
    - Ignore *non-pointer* variables and *non-pointer* fields of objects entirely
- Output: At each program point print the collected facts(shown later).

Implementation Details

- The analysis must be implemented in JAVA Programming language.
- Must use the Soot[1] analysis framework.
- Implement:
    - Kildall's fixed-point Algorithm:
        - Must be independent of the analysis lattice.
        - Transfer functions: LatticeElement $\rightarrow$ LatticeElement where LatticeElement is an Interface. It has methods: *equals*, *join_op*, *tf_assignstmt*, *tf_condstmt* and others that should be implemented for the specific analysis.
    - Points-to Analysis:
        - The *points-to* analysis elements are of LatticeElement type.

[1] https://github.com/soot-oss/soot

Soot Framework Data Structures [2]



[2] https://www.brics.dk/SootGuide/sootsurvivorsguide.pdf

Soot Data Structure for a Method[3]

- Soot represents an input program in several IRs namely, Jimple, Shimple, Baf, Grimp, Dava.
- We will use Jimple IR for analysis.
- Following are the building blocks of a JimpleBody,
  - UnitGraph: Represents a CFG where the nodes are Unit instances and edges may represent normal and exceptional control flows.
  - Unit: A base *interface* used to implement statement kinds in the Jimple IR. Ex. AssignStmt, IfStmt and others.
  - Value: Represent the locals, constants and expressions. Expressions may be BinopExp, InvokeExpr and others.
  - Boxes: References in Soot are called boxes. There are two types – Unitboxes, ValueBoxes. Useful for code transformations.

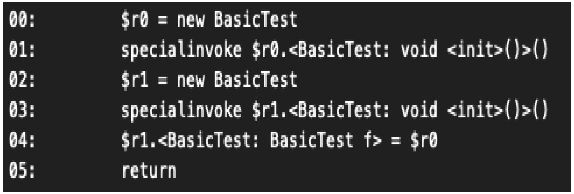
[3] `https://javadoc.io/doc/ca.mcgill.sable/soot/latest/index.html`

Jimple Intermediate Representation - Example1

Java Source:

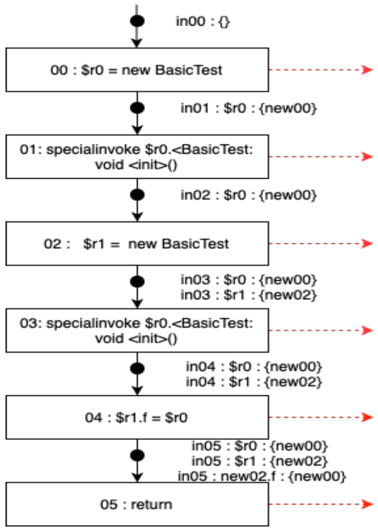
```
class BasicTest
{
 BasicTest f;
 static void fun1()
 {
  BasicTest v1 = new BasicTest();
  BasicTest v2 = new BasicTest();
  v2.f = v1;
 }
}
```

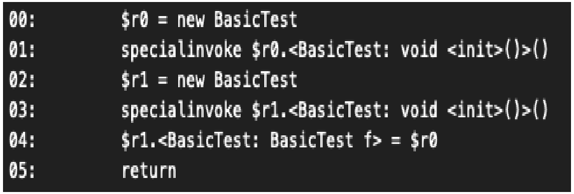
Jimple IR:



```
$r0 = new BasicTest
specialinvoke $r0.<BasicTest: void <init>()>()
$r1 = new BasicTest
specialinvoke $r1.<BasicTest: void <init>()>()
$r1.<BasicTest: BasicTest f> = $r0
return
```

Point-to Dataflow facts at program points





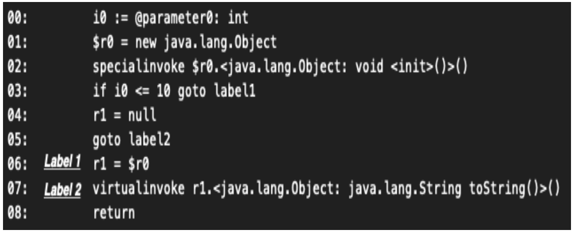
```
00:         $r0 = new BasicTest
01:         specialinvoke $r0.<BasicTest: void <init>()>()
02:         $r1 = new BasicTest
03:         specialinvoke $r1.<BasicTest: void <init>()>()
04:         $r1.<BasicTest: BasicTest f> = $r0
05:         return
```

Jimple Intermediate Representation - Example 2

Java Source:

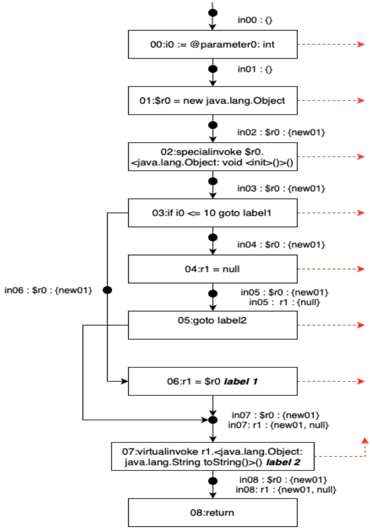
```
class BasicTest
{
 static void foo(int i)
 {
  Object t1 = null;
  Object t2 = new Object();
  Object t3 = null;
  if(i > 10)
  {
    t3 = t1;
  }
  else
  {
    t3 = t2;
  }
    t3.toString();
}
```

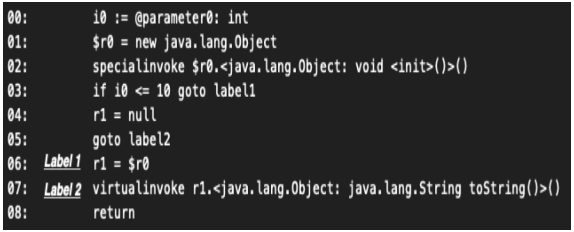
Jimple IR:



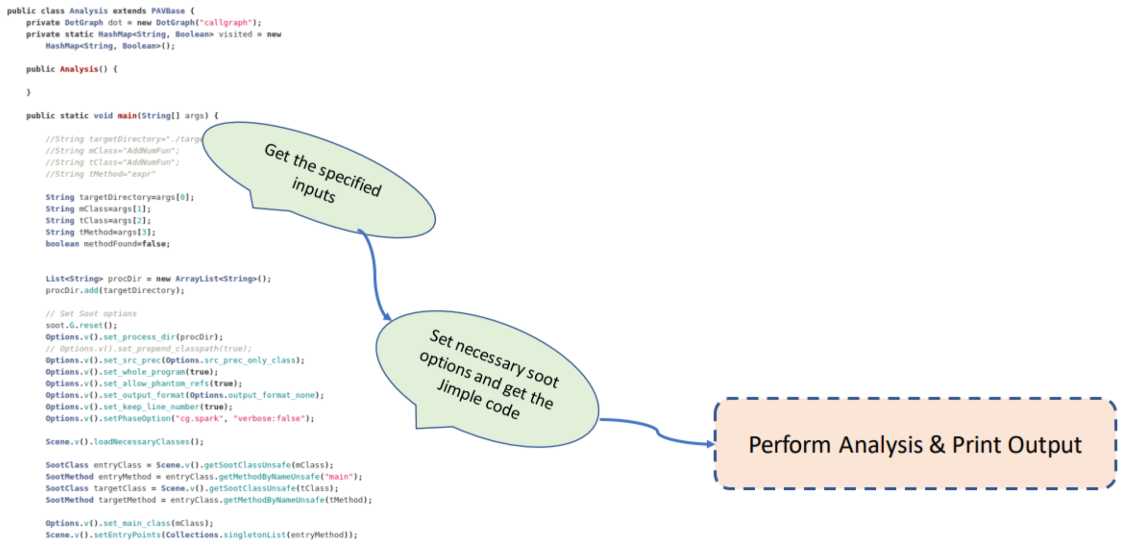
```
00:         i0 := @parameter0: int
01:         $r0 = new java.lang.Object
02:         specialinvoke $r0.<java.lang.Object: void <init>()>()
03:         if i0 <= 10 goto label1
04:         r1 = null
05:         goto label2
06: Label1  r1 = $r0
07: Label2  virtualinvoke r1.<java.lang.Object: java.lang.String toString()>()
08:         return
```

Points-to Dataflow facts at program points




```
00:         i0 := @parameter0: int
01:         $r0 = new java.lang.Object
02:         specialinvoke $r0.<java.lang.Object: void <init>()>()
03:         if i0 <= 10 goto label1
04:         r1 = null
05:         goto label2
06: Label 1 r1 = $r0
07: Label 2 virtualinvoke r1.<java.lang.Object: java.lang.String toString()>()
08:         return
```

Analysis Work Flow - Soot Driver Program



```
public class Analysis extends PAVBase {
    private DotGraph dot = new DotGraph("callgraph");
    private static HashMap<String, Boolean> visited = new
        HashMap<String, Boolean>();

    public Analysis() {

    }

    public static void main(String[] args) {

        //String targetDirectory="./targe
        //String mClass="AddNumFun";
        //String tClass="AddNumFun";
        //String tMethod="expr"

        String targetDirectory=args[0];
        String mClass=args[1];
        String tClass=args[2];
        String tMethod=args[3];
        boolean methodFound=false;


        List<String> procDir = new ArrayList<String>();
        procDir.add(targetDirectory);

        // Set Soot options
        soot.G.reset();
        Options.v().set_process_dir(procDir);
        // Options.v().set_prepend_classpath(true);
        Options.v().set_src_prec(Options.src_prec_only_class);
        Options.v().set_whole_program(true);
        Options.v().set_allow_phantom_refs(true);
        Options.v().set_output_format(Options.output_format_none);
        Options.v().set_keep_line_number(true);
        Options.v().setPhaseOption("cg.spark", "verbose:false");

        Scene.v().loadNecessaryClasses();

        SootClass entryClass = Scene.v().getSootClassUnsafe(mClass);
        SootMethod entryMethod = entryClass.getMethodByNameUnsafe("main");
        SootClass targetClass = Scene.v().getSootClassUnsafe(tClass);
        SootMethod targetMethod = entryClass.getMethodByNameUnsafe(tMethod);

        Options.v().set_main_class(mClass);
        Scene.v().setEntryPoints(Collections.singletonList(entryMethod));
```

Get the specified inputs
Set necessary soot options and get the Jimple code
Perform Analysis & Print Output

LatticeElement Interface

- Kildall implementation should access the dataflow facts as implementation of LatticeElement interface. Furthermore, Kildall implementation should not be tied to specific analysis and should work on any implementation of the LatticeElement.
- v.join(w) returns the join the of the lattice elements pointed to by v and w.
- v.tf_assignment(stmt) returns the result of the applying the transfer function on the input fact pointed to by v using the statement.
- No implementation of methods in this interface should modify the receiver object. Fresh object should be returned. In v.join(w), v, is the receiver object.
- Additional helper interface methods can be added. But you should write about its functionality in README file.

IDE

- You are free to use any IDE for development. (Eg: VS Code, Eclipse, IntelliJ)
- During the demo, your code should work by running the command

  ```
  mvn exec:java
  ```

Expected Output

Your tool should generate two files.

- File 1: should contain the final output of Analysis.
  - Format the output as shown in the next slide.
  - Filename format: class.method.output.txt
    Example: BasicTest.fun1.output.txt
- File 2: should contain the full output.
  - In this file, you should show the updated dataflow fact, at the affected program points, before each step of the Kildall's Algorithm.
  - You can use the same format for output, as shown in the next slide. Each iteration should be separated by a new line.
  - Filename format: class.method.fulloutput.txt
    Example: BasicTest.fun1.fulloutput.txt

Note: Create these files in the same directory as the input .class file.

Expected Output for File 1 - Example 1

Note :"inXX" means incoming facts to statement with label XX.Only text in orange is part of the expected output. foo()

```
BasicTest.fun1: in01: $r0:{new00}
BasicTest.fun1: in02: $r0:{new00}
BasicTest.fun1: in03: $r0: {new00}
BasicTest.fun1: in03: $r1: {new00}
BasicTest.fun1: in04: $r0: {new01}
BasicTest.fun1: in04: $r1: {new00}
BasicTest.fun1: in05: $r0: {new00}
BasicTest.fun1: in05: $r1: {new02}
BasicTest.fun1: in05: new02.f: {new00}
```

Expected Output for File 1 - Example 2

Note :"inXX" means incoming facts to statement with label XX.Only text in orange is part of the expected output. foo()

```
BasicTest.foo: in02: $r0:{new01}
BasicTest.foo: in03: $r0: {new01}
BasicTest.foo: in04: $r0: {new01}
BasicTest.foo: in05: $r0: {new01}
BasicTest.foo: in05: r1: {null}
BasicTest.foo: in06: $r0: {new01}
BasicTest.foo: in07: $r0: {new01}
BasicTest.foo: in07: r1: {new01, null}
BasicTest.foo: in08: $r0: {new01}
BasicTest.foo: in08: r1: {new01, null}
```

Important Notes for your implementation

- Each row in the output should be of format:
  class.method: programpoint: var/object field: {list of symbolic objects}
  Ex: BasicTest.foo: in07: r1: {new01, null}
- The lines in the file are in string sorted order (as shown in the examples). The points-to set of symbolic objects should also be sorted. Refer to the Base.java(fmtOutputLine, fmtOutputData functions) file to understand the right functions to call to print your output.

Important Notes for your implementation

- initial data flow fact (d_0) : An empty points-to list i.e. { } corresponding to each pointer variables and the fields of symbolic objects. In this implementation, $\perp$ is not a distinguished element. Rather, if all variables and all object-fields have empty points-to list, that is $\perp$.
- The transfer functions should not always transmit $\perp$ to $\perp$. For instance, if there is a statement "v = new()", then even if the incoming fact is $\perp$, the outgoing fact v should point to the appropriate newXX. Note: all other statement kinds will transmit $\perp$ as $\perp$ itself.
- At a particular point if a particular pointer-variable or a field of an abstract object has an empty points to list, then its points-to list should not appear in output for that point.

Important Notes for your implementation

- Transfer function for `specialinvoke` statement is *id*.
- Every put-field needs to be weak update. In the statement r1.f = $r0, the r1.f is a put-field and the transfer function for this assignment statement should append to the existing facts of the put-field.
- Each Jimple statement (i.e., Unit) becomes a node in the CFG.
- The point just before each node (Unit) is to be treated as a program point. Hence, program points and nodes correspond one-to-one in this project.
- Program points are to be numbered inXX, where the number for XX is generated using body.getUnits().

Generating CFG

- The generated Jimple IR is missing the "label id" for non linear control flow in the program. You can use cfg to view the labels.
- To generate the cfg for all functions, run the mvn -q clean package exec:java at the root directory, and the cfg for the test cases are seen in the `output/` directory for all functions. The dot is a file format to generate images in the .png file formats.
- If you have Graphviz installed, then the .dot files are rendered as .png images.

Evaluation

What are we looking for ?

- Your tool should not crash.
- Your analysis should be sound.
- Your analysis should be as precise as possible.
- Should not ignore any valid Java constructs (modulo the assumptions stated earlier).

Scoring:

- Each error has an associated penalty.
- Your score: TOTAL SCORE – sum(PENALTIES)

Demo of Phase 1:

- During demo you have to run your tool on predisclosed (public) as well undisclosed (private) testcases.

Phase 2 - Will be updated soon

Evaluation (contd.)

Other Informations:

- Your code will be carefully analyzed with plagiarism checkers.
  - Copying will be dealt with severely.
  - You can learn general Java programming idioms and patterns from other open-source applications, but should not look up Kildall implementation or Pointer Analysis implementation from any source.
  - Don't use any Soot libraries other than the ones already used in `Analysis.java` given to you. Don't use any other libraries, either, other than Java utilities (such as collections).
- Both teammates need to participate. During the demo, we will be evaluating the responses of both members.

Suggestion for Starting

- Understand Soot framework basics, like Units, Values, Boxes.
- Pick one of the public test case targets.
- Traverse the CFG explicitly, and print out the Units and useBoxes, defBoxes, and unitBoxes corresponding to each Unit.

References

- https://noidsirius.medium.com/a-beginners-guide-to-static-program-analysis-using-soot-5aee14a878d
- https://www.sable.mcgill.ca/soot/doc/soot/Unit.html
- https://cs.au.dk/~amoeller/mis/soot.pdf
- https://github.com/soot-oss/soot/wiki/Tutorials
- https://github.com/noidsirius/SootTutorial/tree/master/docs/1

Project Logistics

- Base repo for the project (includes the soot driver):
  - `https://gitlab.com/AlanJojo/2025-pav-assignment`
  - `gitlab.com` will be our online repository hosting service.
- Fork the base repo to your user-account.
  - Fork with repo name as “2025-PAV-FirstName" (eg: 2025-PAV-Deepak)
    - Team number as assigned in the Excel sheet for forming project teams.
  - Create only one fork per team (either of the team member can do this)
    - This will be your “team repo"(aka the "forked repo").
  - Add the other team members to the repo (as Maintainers).
- Add @AlanJojo and @atharv.desai as Maintainers.

Project Logistics (contd.)

- The "forked repo" (your "team repo") will be your common collaboration method. This will enable all team members to work simultaneously on the code.
- Clone the forked repo to your desktop/laptop.
- Follow the instructions in the README.
- Now you may start modifying your implementation.
- After your implementation and testing are completed, upload your codebase as a zip file in MS Teams before the deadline.

Workflow Tips (Not mandatory, but recommended)

- Start the day by doing a “git pull" to receive the changes made by other team members.
    - “git pull" tries to automatically merge changes made by all team members.
    - But if git cannot decide this automatically, a merge conflict will be reported.
    - In that case, you need to resolve it by deciding on how to merge the changes manually.
- “git commit" after making each logical set of changes.
    - logical set is arbitrary and can be as few as 1 line to 100+ lines of change. Ex: After adding a new method/class to the code, or after fixing a bug.
- “git push" at-least once a day.
    - This will enable other team members to see your changes, and will also reduce the chance of merge conflicts.