

Interval Analysis

Course Project Tutorial

E0 227: Program Analysis and Verification 2021

Alvin George and Raseek C

Goal

- Given a Java program, implement a tool that performs Interval analysis.
- Phase 1:
 - Intraprocedural Interval Analysis (using Kildall's algo)
- Phase 2:
 - Interprocedural Interval Analysis. *(details will be specified later)*

Phase1 - Input

- The analyser should work on programs written in Java programming language
- For Phase1, you may assume these restrictions.
 - Only integer variables.
 - No global variables.
 - Only static methods.
 - No Method Calls
- Input format: Input will be a java class file.

Implementation requirements

- The analysis must be **implemented** as a Java Program.
- It must use the *Soot* analysis framework.

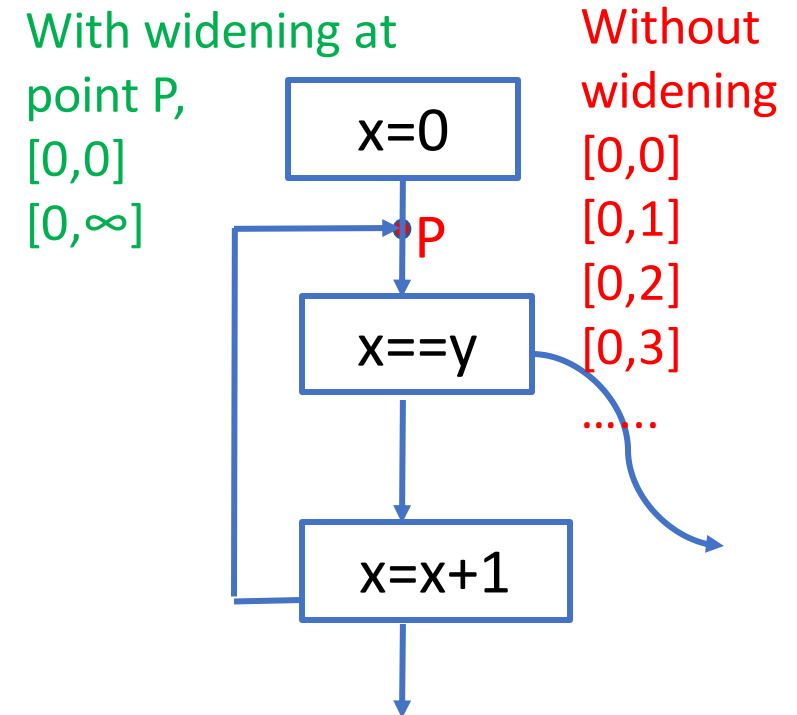
Implementation requirements

- Two parts:
 - Kildall's Algorithm implementation
 - Must be modular. (ie, Algorithm must be agnostic about the particular analysis details)
 - It should assume that transfer functions are of type `LatticeElement -> LatticeElement`
 - Where LatticeElement is an Interface and have these methods
 - `equals`, `join_op`, **`widen_op`**, `tf_assignstmt`, `tf_condstmt`
 - The Specific Analysis (Interval analysis)
 - It must be an implementation of the LatticeElement interface
- Note: The Kildall's Algo, should not directly refer to the IA implementation, instead should access it through the LatticeElement interface methods

Widening Operator (Δ)

- Shorten an ascending infinite chain to a finite height.
- Execution of Kildall's algorithm on Interval Analysis example with infinite height lattice.
 - At P, intervals of x forms an infinite ascending chain.
 - Widening operator makes the chain into a finite one.

- Ref: Patrick Cousot and Radhia Cousot, [Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints](#), In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238--252, New York, NY, USA, 1977. ACM Press. - (**Specifically, Section 9**)



Widening Operator(contd.)

- Widening is done at any point with incoming Loopback edge(s). Other points use join as usual.
- $[i,j] \Delta [k,l] = [\text{if } k < i \text{ then } -\infty \text{ else } i, \text{if } l > j \text{ then } +\infty \text{ else } j]$
- Note, Δ is not symmetric.
 - $[i,j]$ above is the existing value at the point, $[k,l]$ is the incoming value.
- Widening examples
 - $[0,0] \Delta [0,1] = [0,\infty]$
 - $[0,1] \Delta [0,0] = [0,1]$
 - $[0,0] \Delta [-1,0] = [-\infty,0]$

Analysis with Soot Framework

Jimple Intermediate Representation

java source

```
class BasicTest2 {  
    static int add_x(int flag)  
    {  
        int x = 0;  
        int sum = 0;  
        if (flag == 1) {  
            x = x + 10;  
        }  
        sum = sum + x;  
        sum = sum * 3;  
        return sum;  
    }  
}
```

jimple IR

```
<BasicTest2: int add_x(int)>  
    z0 := @parameter0: int  
    b2 = 0  
    if z0 == 0 goto label1  
    b2 = 10  
label1: $i0 = 0 + b2  
        $i1 = $i0 * 3  
        return $i1
```

Jimple IR and CFG

jimple IR

<BasicTest2: **int** add_x(int)>

z0 := @parameter0: **int**

b2 = 0

if z0 == 0 **goto** label1

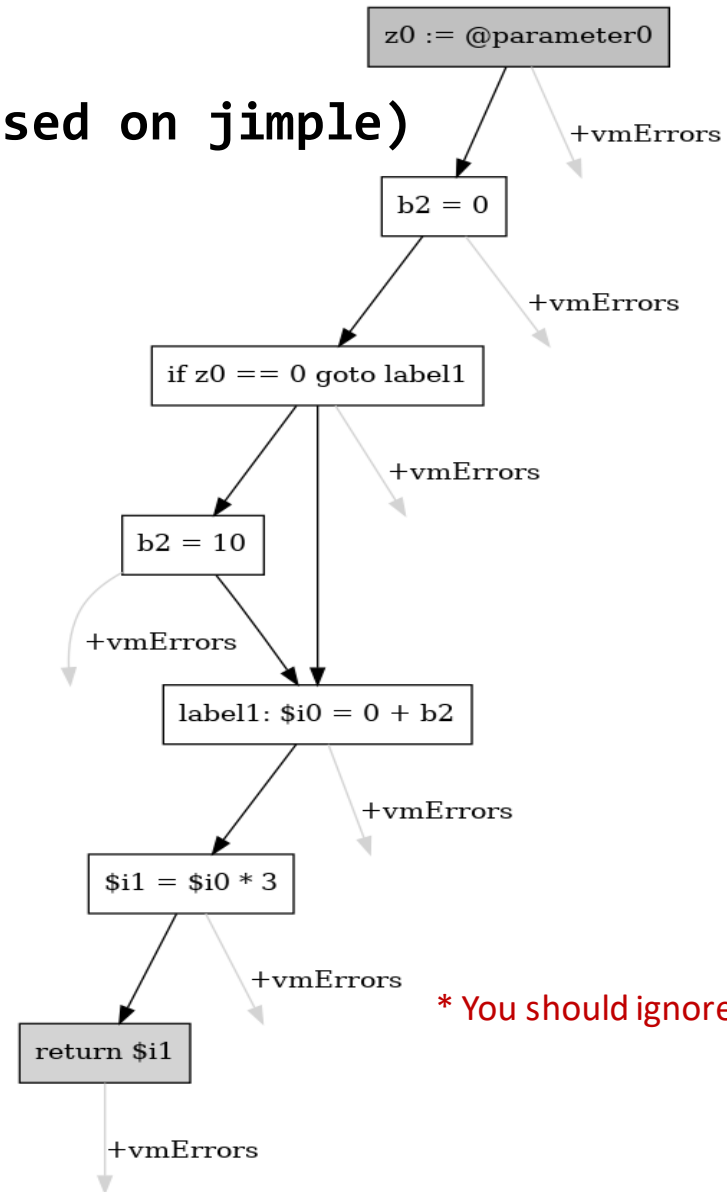
b2 = 10

label1: \$i0 = 0 + b2

\$i1 = \$i0 * 3

return \$i1

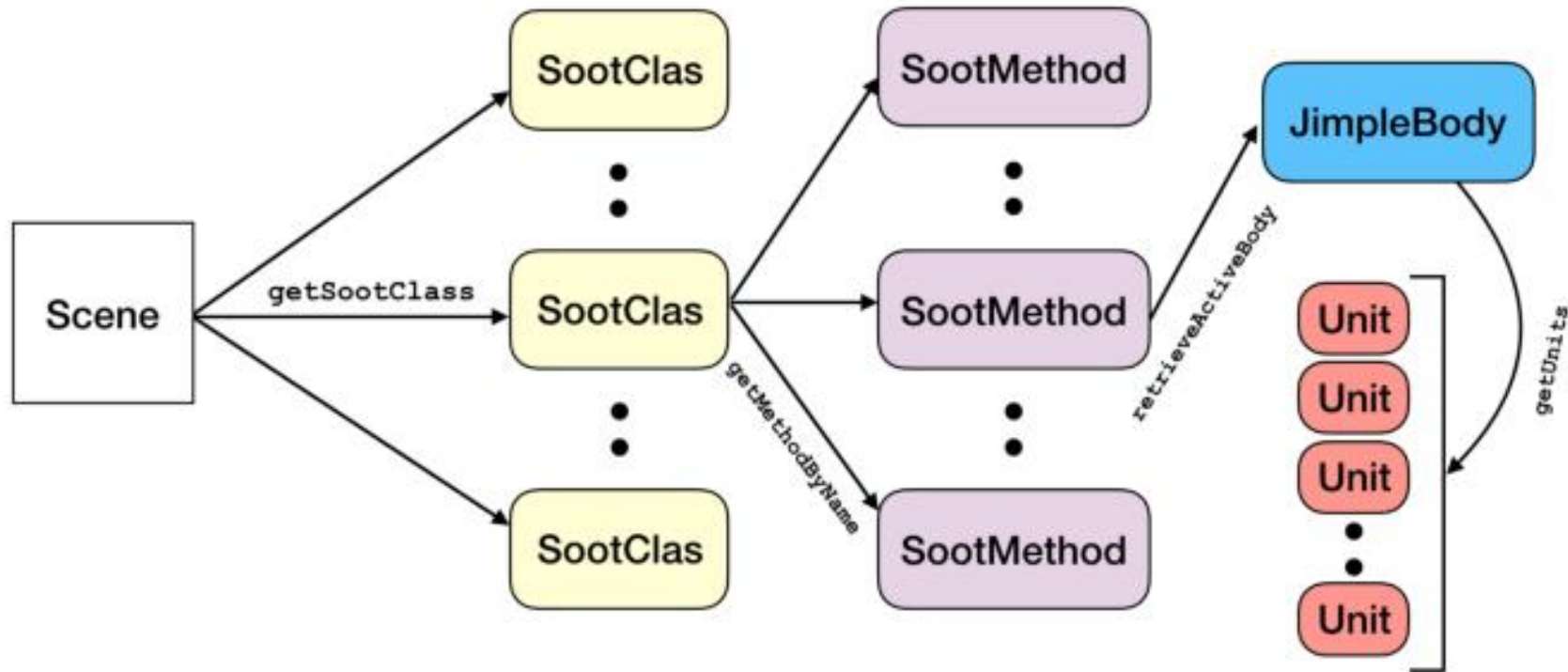
CFG (based on jimple)



* You should ignore exception edges

int add_x(boolean)

Soot Analysis Framework



Soot Analysis Framework(Contd.)

- Units (Stmt, StoreInst, AssignStmt, ...)
- Values (Immediate, FieldRef, AddExpr, ...)
- Boxes (UnitBox, ValueBox, ExprBox, StmtBox, ImmediateBox, VariableBox, ExprBox, ...)

<https://www.sable.mcgill.ca/soot/doc/soot/Unit.html>

Units, Values & Boxes

- Unit

- A code fragment (eg Stmt or Inst), used within Body classes. Intermediate representations must use an implementation of Unit for their code. In general, a unit denotes some sort of unit for execution.

- Value

- Data used as, for instance, arguments to instructions; typical implementations are constants or expressions. Values are typed, clonable and must declare which other Values they use (contain).

- Boxes

- References in Soot are called boxes. There are two types – Unitboxes, ValueBoxes.

UnitGraph

- UnitGraph

- Represents a CFG where the nodes are [Unit](#) instances and edges represent unexceptional and (possibly) exceptional control flow between Units.

Analysis Workflow

Soot Driver Program

- A base soot driver program is provided. (`Analysis.java`)
 - It can be extended for your implementation.
- Input (Program Arguments)
 - `args[0]`: *path to target directory/jar* of classes, where the analysis class resides.
 - `args[1]` : fully qualified name of the main class (entry point of the soot analysis).
 - `args[2]` : fully qualified name of the class containing the method to be analysed.
 - `args[3]` : method name, for which the analysis is to be done.
 - Eg:(1) `~/project/target1 AddNumFun AddNumFun mySum`
 - (2) `~/project/antlr.jar Main ErrorTest findError`

Analysis Workflow – 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="./target";
        //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 not directly refer to IA implementation and should access the dataflow data only via LatticeElement interface.
- Receiver object of the LatticeElement methods should be the existing dataflow fact at a program point. The incoming dataflow fact is passed as a parameter for widen/join operator.
- No implementation of methods in this interface should modify the receiver object.
 - Fresh object should be returned.

IDE

- You are free to use any IDE for development. (Eg: Eclipse, IntelliJ)
- Arguments to Analysis.java can be given in the Run Configuration.
- During the demo, your code should work by calling Analysis.java with command line arguments from terminal.

Expected Output

- You should generate two files.
 - **File1** – should contain the *final output* of Analysis.
 - Format the output as shown in the *next slide*.
 - Filename format: `class.method.output.txt`
Eg: `BasicTest1.myIncrement.output.txt`
 - **File2** – should contain the *full output* (including intermediate outputs).
 - In this file, you should show the updated dataflow fact, at the affected program points, after each step of the Kildall's Algorithm.
 - You can use the same format for output, as shown in the next slide.
 - Filename format: `class.method.fulloutput.txt`
Eg: `BasicTest1.myIncrement.fulloutput.txt`

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

Expected Output for File 1

tc01-myIncrement()

```
BasicTest1.myIncrement: in00: $i1: [-inf, +inf]
BasicTest1.myIncrement: in00: @parameter0: [-inf, +inf]
BasicTest1.myIncrement: in00: i0: [-inf, +inf]
BasicTest1.myIncrement: in01: $i1: [-inf, +inf]
BasicTest1.myIncrement: in01: @parameter0: [-inf, +inf]
BasicTest1.myIncrement: in01: i0: [-inf, +inf]
BasicTest1.myIncrement: in02: $i1: [-inf, +inf]
BasicTest1.myIncrement: in02: @parameter0: [-inf, +inf]
```

Expected Output for File 1

tc01-mySum()

```
BasicTest1.mySum: in00: i0: [-inf, +inf]
BasicTest1.mySum: in00: i1: [-inf, +inf]
BasicTest1.mySum: in01: i0: [0, 0]
BasicTest1.mySum: in01: i1: [-inf, +inf]
BasicTest1.mySum: in02: i0: [0, +inf]
BasicTest1.mySum: in02: i1: [0, +inf]
BasicTest1.mySum: in03: i0: [0, +inf]
BasicTest1.mySum: in03: i1: [0, 10]
BasicTest1.mySum: in04: i0: [0, +inf]
BasicTest1.mySum: in04: i1: [0, 10]
BasicTest1.mySum: in05: i0: [0, +inf]
BasicTest1.mySum: in05: i1: [1, 11]
BasicTest1.mySum: in06: i0: [0, +inf]
BasicTest1.mySum: in06: i1: [11, +inf]
```

Expected Output Format of File 1

- Each row in the output should be of format:
`class.method: programpoint: var: [lower, upper]`
- Format for the Result corresponding to interval `[lower, upper]`
 - `"-inf"` if lower is $-\infty$, else `lower.toString()`
 - `"+inf"` if upper is $+\infty$, else `upper.toString()`
- The lines in the file are in string sorted order (as shown in the examples).
- If dataflow fact at a particular point is `\bot`, then no output lines should appear at that point.

Expected Output Format of File 1

- 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 (unlike in the theory).
- Program points are to be numbered in00, in01, in02, etc.
 - The numbering is in the order as returned by `body.getUnits()`
 - At each program point, **ALL variables** in the method under analysis, need to be shown.
- Initial data flow fact (at program entry) is `[-inf, +inf]` for all variables.

Other Important Information

Evaluation

- What we are 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: $\text{TOTAL SCORE} - \text{sum(PENALTIES)}$

Evaluation (Contd.)

- Phase 1 Submission Deadline:
 - Date: October 29th, 2021.
 - Extended to: November 3rd, 2021 (11:59pm)
- Demo of Phase 1:
 - During demo: run your tool on predisclosed (public) as well undisclosed (private) testcases.
- You should add your own testcases, for increased test coverage.
- Credits will be divided between Phase I and Phase II.
- No changes to the score of Phase I shall be entertained after the demo of Phase I.

Also,

- 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 Interval 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.
 - Ideally, we would like to see the commits of both members. Nevertheless, this is not strictly enforced.

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 (Soot)

- A Survivors Guide to Soot
 - <https://cs.au.dk/~amoeller/mis/soot.pdf>
- Soot Tutorials
 - <https://github.com/soot-oss/soot/wiki/Tutorials>
 - <https://github.com/noidsirius/SootTutorial/tree/master/docs/1>
 - <https://noidsirius.medium.com/a-beginners-guide-to-static-program-analysis-using-soot-5aee14a878d>

Project Logistics

- Base repo for the project (includes the soot driver)
 - <https://gitlab.com/alvg/pav-2021-project-base>
 - gitlab.com will be our online repository hosting service.
- Create an account in gitlab.com
 - Each member of the team should have a gitlab account.
- Fork the base repo to your useraccount
 - Fork with repo name as "pav2021-teamXX" (eg: pav2021-team08)
 - 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 @alvg, @raseekc as Maintainers

Project Logistics

- 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

Project Logistics

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.
 - Eg: 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.

Phase2

Phase 2 Requirements

- The Phase 2 project will be about extending your Phase 1 implementation to make it inter-procedurally context-sensitive using **"approximate" call-strings approach**.
- You are to use a fixed **call-string length bound** of 1.
- You need to extend your Kildall implementation to use call-strings.
 - As in Phase 1,
 - the Kildall implementation (incorporating call-strings) should not be specific to Interval Analysis,
 - and should be capable of working on arbitrary lattices that are implemented by extending LatticeElement.
- Your Analysis should also handle recursion

Assumptions on the input

- All variables are integer variables,
- All global variables are static
 - Methods might have local variables.
- Only static methods
- No arguments to methods
- No return values from methods, except for the special method `iiscPavUtil.random()` which may be declared as shown below in the benchmark program being analyzed:

```
public class iiscPavUtil {  
    | | | public static int random();  
}
```

You should not analyze the "random()" method, instead should treat it as returning an arbitrary integer.

- The same set of integer operations as used in Phase 1.

Tool Interface

- The interface to run your tool should be the same as before
 - a **target class** name and a **target method** name should be accepted as argument.
 - However, all functions that are called directly or transitively by this target method should be analyzed as well
- The expected output from the Tool is in two text files (as in phase1).

Public Testcase example

```
1  class PubTest5 {  
2      static int randInt;  
3      static int a;  
4      static void main() {  
5          randInt = iiscPavUtil.random();  
6          if (randInt == 1) {  
7              a = 1;  
8              foo();  
9          }  
10         else {  
11             a = 2;  
12             foo();  
13         }  
14     }
```

```
15     static void foo() {  
16         if (a == 1) {  
17             bar();  
18             return;  
19         }  
20         else {  
21             bar();  
22             return;  
23         }  
24     }  
25     static void bar() {  
26         a = a + 1;  
27     }  
28 }
```

Example public testcase

Corresponding (hand-annotated) Jimple code [excerpt]

```
static void main()
{
in00    int $i0, $i1;
in01    $i0 = staticinvoke <iiscPavUtil: int random()>(); // S1
in02    <PubTest5: int randInt> = $i0;
in03    $i1 = <PubTest5: int randInt>;
in04    if $i1 != 1 goto label1;
in05    <PubTest5: int a> = 1;
in06    staticinvoke <PubTest5: void foo()>();           // S2
in07    goto label2;
in08 label1:
| | | | <PubTest5: int a> = 2;
in09    staticinvoke <PubTest5: void foo()>();           // S3
in10 label2:
| | | | return;
}
```

```
static void foo()
{
in00    int $i0;
in01    $i0 = <PubTest5: int a>;
in02    if $i0 != 1 goto label1;
in03    staticinvoke <PubTest5: void bar()>();           // S4
in04    return;
in05 label1:
| | | | staticinvoke <PubTest5: void bar()>();           // S5
in06    return;
}
static void bar()
{
in00    int $i0, $i1;
in01    $i0 = <PubTest5: int a>;
in02    $i1 = $i0 + 1;
in03    <PubTest5: int a> = $i1;
in04    return;
}
```

Naming Program Points, Call-sites

- Program point numbering is same as in phase 1
- Call-site names for use in call string should use "`classname.methodname.inXX`" format.
 - Eg: Call-site name corresponding to program point "`in03`" in the "`foo()`" method of class "`PubTest5`" will become "`PubTest5.foo.in03`"
 - Use "`@`" to denote the empty call-string.
- In the output, for each program point, your tool should emit all integer variables including intermediate variables. (i.e, all global variables, and all local variables of the method containing that program point).
 - Global variable "`var1`" in "`class1`" should be named as "`class1.var1`"
 - (Recall that all global variables are static).

Expected output format (for file1)

Sample output for target-class: "PubTest5" and target-method: "main()"

```
PubTest5.bar: in00: PubTest5.foo.in03 => $i0: [-inf, +inf]
PubTest5.bar: in00: PubTest5.foo.in03 => $i1: [-inf, +inf]
PubTest5.bar: in00: PubTest5.foo.in03 => PubTest5.a: [1, 1]
PubTest5.bar: in00: PubTest5.foo.in03 => PubTest5.randInt: [-inf, +inf]
PubTest5.bar: in00: PubTest5.foo.in05 => $i0: [-inf, +inf]
PubTest5.bar: in00: PubTest5.foo.in05 => $i1: [-inf, +inf]
PubTest5.bar: in00: PubTest5.foo.in05 => PubTest5.a: [2, 2]
PubTest5.bar: in00: PubTest5.foo.in05 => PubTest5.randInt: [-inf, +inf]
...
PubTest5.foo: in02: PubTest5.main.in06 => $i0: [1, 1]
PubTest5.foo: in02: PubTest5.main.in06 => PubTest5.a: [1, 1]
PubTest5.foo: in02: PubTest5.main.in06 => PubTest5.randInt: [-inf, +inf]
PubTest5.foo: in02: PubTest5.main.in09 => $i0: [2, 2]
PubTest5.foo: in02: PubTest5.main.in09 => PubTest5.a: [2, 2]
PubTest5.foo: in02: PubTest5.main.in09 => PubTest5.randInt: [-inf, +inf]
PubTest5.foo: in03: PubTest5.main.in06 => $i0: [1, 1]
...
PubTest5.main: in00: @ => $i0: [-inf, +inf]
PubTest5.main: in00: @ => $i1: [-inf, +inf]
PubTest5.main: in00: @ => PubTest5.a: [-inf, +inf]
PubTest5.main: in00: @ => PubTest5.randInt: [-inf, +inf]
```

You can use the same format for the File2 as well.

*For the above public test case, the complete version of expected output is provided in the gitlab repo as a text file, alongside the source code. (see target4-pub/PubTest5.main.exp-out.txt)

Thank You !