Null Dereference Analysis

E0-227 : Program Analysis and Verification, 2015

Aravind Acharya, Shyam S

Instructors: Prof. K. V. Raghavan, Prof. D. D’Souza

Computer Science and Automation,

Indian Institute of Science, Bangalore

Goal

Given a Java program, to implement a tool that performs interprocedural pointer analysis on a user-specified method using Iterative Method.

The Programming Language

- Your analysis should work on any *arbitrary* Java program. All syntactic features of Java ought to be handled with assumptions which are stated at a later slide.

Pointer Analysis (An Introduction)

Gives information about what objects are pointed to by which pointer variables

```
void foo {
1:  temp1=null;
2:  temp2=new();
3:  temp3= temp2;
}
```

temp1 points to null

temp2 points to obj created at site s2

temp3 points to obj pointed to by temp2

1: temp1-> {NULL}

2: temp1-> {NULL},
   temp2->{O2}

3: temp1-> {NULL},
   temp2->{O2},
   temp3->{O2}

O2 represents all symbolic objects created at line 2

Example 2

```
1: void foo(){
2:    temp1=new();
3:    if(i%2==0){ //Assume i to be global
random integer.
4:       temp1=new();
5:    }
6:    temp2=temp1;
// temp2 may point to both O2 and O4
here.
7: }
```

2: temp1->{O2}

4: temp1->{O4}

5: temp1->{O2,O4}

6: temp1->{O2,O4}, temp2->{O2, O4}

Example 3 (interprocedural case)

```
/* consider temp1, temp2 are global
 and do not point to anything yet
 */
2:    void foo(){
3:        temp1 = null;
4:        bar();
5:        temp2=temp1;
6:    }
7:    void bar() {
8:        temp1=new();
9:        return;
10:   }
```

Analysis of method foo:

3: temp1->{NULL}

5: temp1->{O8},temp2->{O8}

8: temp1->{O8}

Implementation Details

WALA

- IBM T. J. WAtson Libraries for Analysis
- Framework for analysis of Java programs
- WALA, like other tools, converts Java byte code to an “intermediate representation”, which is more amenable to analysis
- Your analysis take an Intermediate Representation (IR) as the input.

Sample 1 – WALA Intermediate Representation

Static Single Assignment (SSA)

```
1: public class SampleTests {
2:
3:  public static void foo(int i) {
4:      SampleTests t1 = new SampleTests();
5:      SampleTests t2 = new SampleTests();
6:      SampleTests t3 = null;
7:      if(i>10) {
8:          t3 = t1;
9:      } else {
10:             t3 = t2;
11:     }
12:     t3.toString();
13: }
```

```
BB0
BB1
0   v3 = new <Application,LTestCases/SampleTests>@0(line 4)
BB2
2   invokespecial < Application, LTestCases/SampleTests, <init>()V > v3 @4
exception:v4(line 4)
BB3
4   v5 = new <Application,LTestCases/SampleTests>@8(line 5)
BB4
6   invokespecial < Application, LTestCases/SampleTests, <init>()V > v5 @12
exception:v6(line 5)
BB5
12   conditional branch(le, to iindex=16) v1,v8:#10(line 7) {1=[i]}
BB6
15   goto (from iindex= 15 to iindex = 18)   (line 9)
BB7
BB8
        v9 = phi  v3,v5
19   v11 = invokevirtual < Application, Ljava/lang/Object, toString()
Ljava/lang/String; > v9 @32 exception:v10(line 12) {9=[t3]}
BB9
21   return                              (line 13)
BB10
```

Sample 1 – WALA Intermediate Representation

Control Flow Graph Nodes

```
1: public class SampleTests {
2:
3:  public static void foo(int i) {
4:      SampleTests t1 = new SampleTests();
5:      SampleTests t2 = new SampleTests();
6:      SampleTests t3 = null;
7:      if(i>10) {
8:          t3 = t1;
9:      } else {
10:             t3 = t2;
11:     }
12:     t3.toString();
13: }
```

```
BB0
BB1
0   v3 = new <Application,LTestCases/SampleTests>@0(line 4)
BB2
2   invokespecial < Application, LTestCases/SampleTests, <init>()V > v3 @4
exception:v4(line 4)
BB3
4   v5 = new <Application,LTestCases/SampleTests>@8(line 5)
BB4
6   invokespecial < Application, LTestCases/SampleTests, <init>()V > v5 @12
exception:v6(line 5)
BB5
12   conditional branch(le, to iindex=16) v1,v8:#10(line 7) {1=[i]}
BB6
15   goto (from iindex= 15 to iindex = 18)   (line 9)
BB7
BB8
        v9 = phi  v3,v5
19   v11 = invokevirtual < Application, Ljava/lang/Object, toString()
Ljava/lang/String; > v9 @32 exception:v10(line12) {9=[t3]}
BB9
21   return                          (line 13)
BB10
```

Sample 1 – WALA Intermediate Representation

Phi Nodes

```
1: public class SampleTests {
2:
3:  public static void foo(int i) {
4:      SampleTests t1 = new SampleTests();
5:      SampleTests t2 = new SampleTests();
6:      SampleTests t3 = null;
7:      if(i>10) {
8:          t3 = t1;
9:      } else {
10:             t3 = t2;
11:     }
12:     t3.toString();
13:  }
```

```
BB0
BB1
0   v3 = new <Application,LTestCases/SampleTests>@0(line 4)
BB2
2   invokespecial < Application, LTestCases/SampleTests, <init>()V > v3 @4
exception:v4(line 4)
BB3
4   v5 = new <Application,LTestCases/SampleTests>@8(line 5)
BB4
6   invokespecial < Application, LTestCases/SampleTests, <init>()V > v5 @12
exception:v6(line 5)
BB5
12   conditional branch(le, to iindex=16) v1,v8:#10(line 7) {1=[i]}
BB6
15   goto (from iindex= 15 to iindex = 18)   (line 9)
BB7
BB8
         v9 = phi  v3,v5
19   v11 = invokevirtual < Application, Ljava/lang/Object, toString()
Ljava/lang/String; > v9 @32 exception:v10(line 12) {9=[t3]}
BB9
21   return                          (line 13)
BB10
```

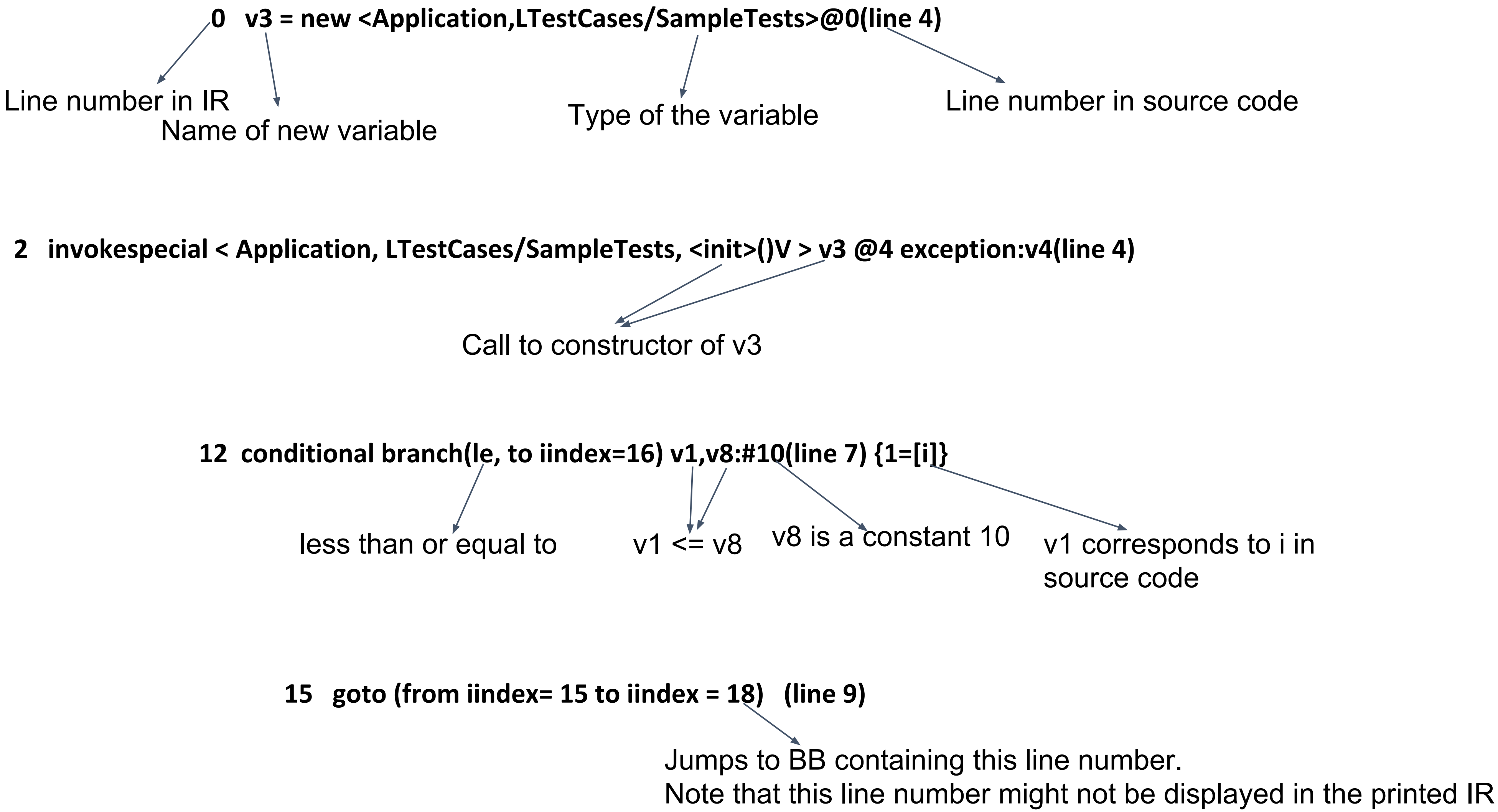
Sample 1 – WALA CFG

```java
public class SampleTests {

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

```
BB0[-1..-2]
   -> BB1
BB1[0..0]
   -> BB2
   -> BB10
BB2[1..2]
   -> BB3
   -> BB10
BB3[3..4]
   -> BB4
   -> BB10
BB4[5..6]
   -> BB5
   -> BB10
BB5[7..12]
   -> BB7
   -> BB6
BB6[13..15]
   -> BB8
BB7[16..17]
   -> BB8
BB8[18..19]
   -> BB9
   -> BB10
BB9[20..21]
   -> BB10
BB10[-1..-2]
```

Understanding WALA IR: Statements



```
0   v3 = new <Application,LTestCases/SampleTests>@0(line 4)
```

Line number in IR

Name of new variable

Type of the variable

Line number in source code

```
2   invokespecial < Application, LTestCases/SampleTests, <init>()V > v3 @4 exception:v4(line 4)
```

Call to constructor of v3

```
12  conditional branch(le, to iindex=16) v1,v8:#10(line 7) {1=[i]}
```

less than or equal to

v1 <= v8

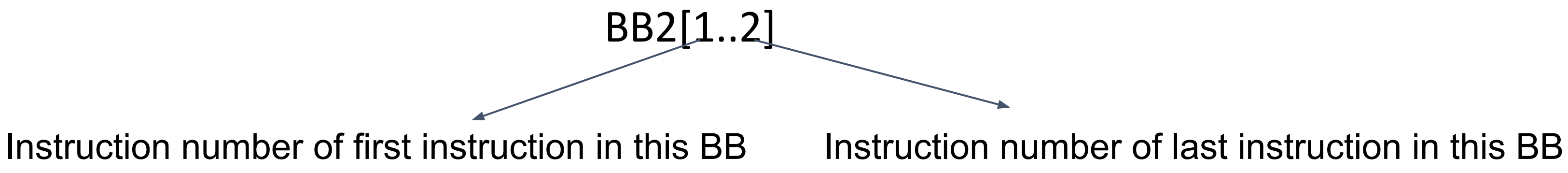
v8 is a constant 10

v1 corresponds to i in source code

```
15   goto (from iindex= 15 to iindex = 18)   (line 9)
```

Jumps to BB containing this line number.
Note that this line number might not be displayed in the printed IR

Understanding WALA IR: Basic Blocks



BB2[1..2]

Instruction number of first instruction in this BB

Instruction number of last instruction in this BB

Note: These instructions might not be visible in the IR

If a BB contains multiple outgoing edges and one of it is an outgoing edge to the last BB then there is a high possibility that that edge is due to a possible exception that can be thrown. You can check which edges are exception edges from the CFG using WALA APIs (SSACFG.hasExceptionalEdge and SSACFG. getExceptionalSuccessors)

```
BB1[0..0]
   -> BB2
   -> BB10
```

```
BB1
0   v3 = new <Application,LTestCases/SampleTests>@0(line 4)
```

For this example, BB1 has an outgoing edge to BB2 and BB10. The edge BB1 -> BB10 corresponds to the exception that is thrown when a new object is allocated (if allocation fails due to any reason).

Understanding WALA IR: Phi nodes and conditionals

```
BB5[7..12]
   -> BB7
   -> BB6
```

```
BB5
12   conditional branch(le, to iindex=16) v1,v8:#10(line 7) {1=[i]}
```

Outgoing edge of BB5 corresponding to the true branch of the condition goes to BB7 and the false branch to BB6

```
BB6[13..15]
   -> BB8
BB7[16..17]
   -> BB8
BB8[18..19]
   -> BB9
   -> BB10
```

```
BB6
15   goto (from iindex= 15 to iindex = 18)   (line 9)
BB7
BB8
         v9 = phi  v3,v5
19   v11 = invokevirtual < Application, Ljava/lang/Object, toString()Ljava/lang/String; > v9 @32 exception:v10(line 12) {9=[t3]}
```

From the CFG it is clear that BB8 is the merge node of the conditional in BB5. Thus it contains the phi instruction. In the phi instruction, v3 corresponds to the value that comes from the false branch of BB5 (i.e. BB6) and v5 to the true branch (i.e. BB7). This is in reverse with respect to the order shown in the CFG for BB5. Thus if BB5 has a deterministic conditional, then you need to be careful as to which variable v9 should be assigned to.

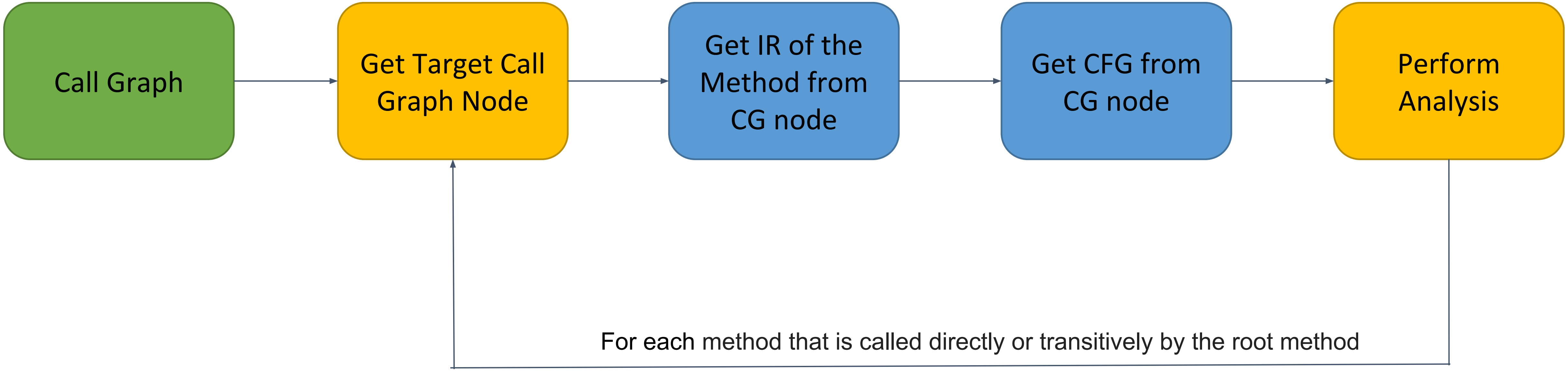
Analysis Workflow – Setting Up

- Call graph – A directed graph in which nodes are the subroutines and an edge from node n_1 to node n_2 represents that n_1 calls n_2
- Initial Set-up –
  - Code provided to set up the analysis
  - Class – SetUpAnalysis.java
  - Call graph of the entire JAR and IR of the method to be analysed are provided
  - printnodes() – prints the nodes in the call graph
  - printIR() - prints the IR of the given method
- No need to change the provided code

Analysis Workflow – Providing Arguments

- Main Class: PAVPointerAnalysis.java
- Input (Program Arguments)
  - args[0]: path to the application jar
  - args[1]: fully qualified name of the main class. Fully qualified name has the format L<package_name>/<class_name>
  - args[2]: fully qualified name of the class containing the method to be analyzed
  - args[3]: root method to be analyzed
- E.g. the arguments toanalyse foo in Sample 1 would be:
  <path_to_jar> LTestCases/SampleTests LTestCases/SampleTests foo(I)V

Analysis Workflow – What you need to do


Call Graph
Get Target Call Graph Node
Get IR of the Method from CG node
Get CFG from CG node
Perform Analysis
For each method that is called directly or transitively by the root method

Analysis

- Need to analyze each called function completely, irrespective of whether it returns a reference or not
    - Only "`new`" calls should be treated as allocation sites
- Use the **iterative** approach for the analysis
- Information to be maintained at each program point – table of points-to-graphs
- Your analysis can be restricted to the portion of the program that includes the given root method and the methods called (directly or transitively) by it, excluding library methods.

Analysis Workflow – What You Need To Do

- Devise a suitable data structure for the points-to graph
- For computing the points-to sets,
  - ➢ you need to maintain a table of points-to graph at every program point in the CFG
  - ➢ you should design the lattice as well as design the transfer functions yourself
- For Interprocedural Analysis,
  - ➢ use the iterative method.
  - ➢ design the transfer functions for the call and return nodes.
- The analysis needs to be flow sensitive.
- Write the output to a file in the specified format
- The requirements for Phase II will be specified later

Assumptions for Phase 1.

- All fields of all user-defined objects are of primitive type.
- No user-defined variables are static. There will be no uninitialized variables.
- All `if` statements other than `if(var == null)` and `if(var != null)` must be considered as non-deterministic.
- No user-defined arrays.
- Methods will have parameters and return values.
- Library methods will be present but can be ignored. (can be treated as a no-op).
- Do not propagate dataflow facts through exception edges. Act as if they don't exist.
- Try catch blocks will not be present in any of the programs.
- Static and non-static methods have to be handled (non-static methods have v1 as ‘this’).
- Recursive functions need to be handled.

Analysis Workflow – The Classes

```
public class PAVPointerAnalysis {

        private SetUpAnalysis setup;

        public PAVPointerAnalysis(String classpath, String mainClass, String analysisClass, String analysisMethod) {
                setup = new SetUpAnalysis(classpath, mainClass, analysisClass, analysisMethod);
        }

        public static void main(String[] args) throws Exception {
                String classpath, mainClass, analysisClass, analysisMethod;

                classpath = args[0];
                mainClass = args[1];
                analysisClass = args[2];
                analysisMethod = args[3];

                PAVPointerAnalysis pAnalysis = new PAVPointerAnalysis(classpath, mainClass, analysisClass, analysisMethod);
                pAnalysis.runAnalysis();

        }
}
```

Sample 1 – Input Code with IR

arguments: <path_to_jar> LTestCases/SampleTests LTestCases/SampleTests foo(I)V

```
1: public class SampleTests {
2:
3:  public static void foo(int i) {
4:       SampleTests t1 = new SampleTests();
5:       SampleTests t2 = new SampleTests();
6:       SampleTests t3 = null;
7:       if(i>10) {
8:            t3 = t1;
9:       } else {
10:           t3 = t2;
11:      }
12:      t3.toString();
13:  }
```

```
BB0
BB1
0   v3 = new <Application,LTestCases/SampleTests>@0(line 6)
BB2
2   invokespecial < Application, LTestCases/SampleTests, <init>()V > v3 @4
exception:v4(line 6)
BB3
4   v5 = new <Application,LTestCases/SampleTests>@8(line 7)
BB4
6   invokespecial < Application, LTestCases/SampleTests, <init>()V > v5 @12
exception:v6(line 7)
BB5
12   conditional branch(le, to iindex=16) v1,v8:#10(line 9) {1=[i]}
BB6
15   goto (from iindex= 15 to iindex = 18)   (line 11)
BB7
BB8
         v9 = phi  v3,v5
19   v11 = invokevirtual < Application, Ljava/lang/Object, toString()
Ljava/lang/String; > v9 @32 exception:v10(line 14) {9=[t3]}
BB9
21   return                              (line 15)
BB10
```

Sample 1 – Expected Output

```
BB0
BB1
0   v3 = new <Application,LTestCases/SampleTests>@0(line 6)
BB2
2   invokespecial < Application, LTestCases/SampleTests, <init>()V > v3 @4
exception:v4(line 6)
BB3
4   v5 = new <Application,LTestCases/SampleTests>@8(line 7)
BB4
6   invokespecial < Application, LTestCases/SampleTests, <init>()V > v5 @12
exception:v6(line 7)
BB5
12   conditional branch(le, to iindex=16) v1,v8:#10(line 9) {1=[i]}
BB6
15   goto (from iindex= 15 to iindex = 18)   (line 11)
BB7
BB8
          v9 = phi  v3,v5
19   v11 = invokevirtual < Application, Ljava/lang/Object, toString()
Ljava/lang/String; > v9 @32 exception:v10(line 14) {9=[t3]}
BB9
21   return                              (line 15)
BB10
```

```
BB0 -> BB1:
BB1 -> BB2: {(v3 ->{new0})}
BB2 -> BB3: {(v3 ->{new0})}
BB3 -> BB4: {(v3 ->{new0}), (v5->{new4})}
BB4 -> BB5: {(v3 ->{new0}), (v5->{new4})}
BB5 -> BB6: {(v3 ->{new0}), (v5->{new4})}
BB5 -> BB7: {(v3 ->{new0}), (v5->{new4})}
BB6 -> BB8: {(v3 ->{new0}), (v5->{new4})}
BB7 -> BB8: {(v3 ->{new0}), (v5->{new4})}
BB8 -> BB9: {(v3 ->{new0}), (v5->{new4}), (v9->{new0,
new4})}
BB9 -> BB10: {(v3 ->{new0}), (v5->{new4}), (v9->
{new0, new4})}
```

Sample 2 - Input Code with IR

arguments: <path_to_jar> LTestCases/SampleTests LTestCases/SampleTests main([Ljava/lang/String;)V

```
3:  public class SampleTests {
17:  public static void bar(SampleTests t1) {
18:     SampleTests t2 = new SampleTests();
19:     SampleTests t3 = null;
20:     if(t1==null) {
21:         t3 = t2;
22:     } else {
23:         t3 = t1;
24:     }
25:     t3.toString();
26:  }
27:
28:  public static void main(String[] args) {
29:     bar(null);
30:  }
31: }
```

main:

```
BB0
BB1
1   invokestatic < Application, LTestCases/SampleTests, bar
(LTestCases/SampleTests;)V > v3:#null @1 exception:v4(line 29)
BB2
2   return                              (line 30)
BB3
```

bar:

```
BB0
BB1
0   v3 = new <Application,LTestCases/SampleTests>@0(line 18)
BB2
2   invokespecial < Application, LTestCases/SampleTests, <init>()V > v3 @4
exception:v4(line 18)
BB3
8   conditional branch(ne, to iindex=12) v1,v5:#null(line 20) {1=[t1], 5=[t3]}
BB4
11   goto (from iindex= 11 to iindex = 14)   (line 22)
BB5
BB6
        v6 = phi  v3,v1
15   v8 = invokevirtual < Application, Ljava/lang/Object, toString()
Ljava/lang/String; > v6 @22 exception:v7(line 25) {6=[t3]}
BB7
17   return                             (line 26)
BB8
```

Sample 2 - Expected Output

```
main:
BB0
BB1
1   invokestatic < Application, LTestCases/SampleTests, bar
(LTestCases/SampleTests;)V > v3:#null @1 exception:v4(line 29)
BB2
2   return                              (line 30)
BB3

bar:
BB0
BB1
0   v3 = new <Application,LTestCases/SampleTests>@0(line 18)
BB2
2   invokespecial < Application, LTestCases/SampleTests, <init>()V > v3 @4
exception:v4(line 18)
BB3
8   conditional branch(ne, to iindex=12) v1,v5:#null(line 20) {1=[t1], 5=[t3]}
BB4
11   goto (from iindex= 11 to iindex = 14)   (line 22)
BB5
BB6
        v6 = phi  v3,v1
15   v8 = invokevirtual < Application, Ljava/lang/Object, toString()
Ljava/lang/String; > v6 @22 exception:v7(line 25) {6=[t3]}
BB7
17   return                              (line 26)
BB8
```

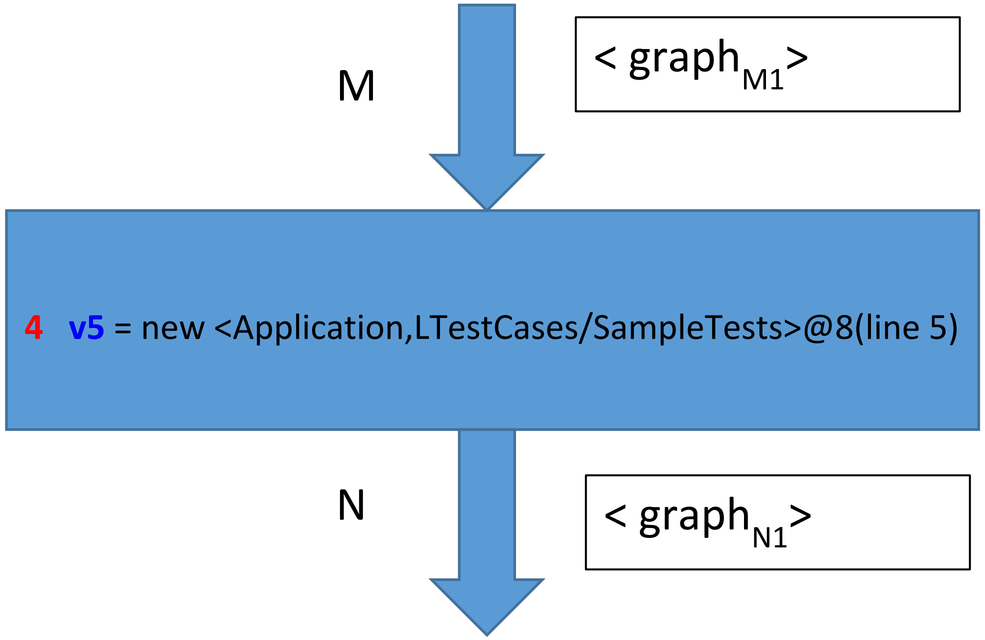
```
main:
BB0 -> BB1:
BB1 -> BB2:     {(v3->{null})}
BB2 -> BB3:     {(v3->{null})}
bar:
BB0 -> BB1:     {(v1 -> {null})}
BB1 -> BB2:     {(v1 -> {null}),
                 (v3 -> {new0})}
BB2 -> BB3:     {(v1 -> {null}),
                 (v3 -> {new0})}
BB3 -> BB5:     {(v1 -> {null}),
                 (v3 -> {new0})}
BB3 -> BB4:
BB5 -> BB6:     {(v1 -> {null}),
                 (v3 -> {new0})}
BB4 -> BB7:
BB6 -> BB7:     {(v1 -> {null}),
                (v3 -> {new0}),
                (v6 -> {new0})}
BB7 -> BB8:     {(v1 -> {null}),
                 (v3 -> {new0}),
                 (v6 -> {new0})}
```

If the flag is not set, show joined results at each program point, as depicted in this slide. In both cases, show only the final fix-point values.

The analysis decides that one of the branches from the conditional is never taken and hence the result is precise.

Naming Convention

- Maintaining the information of the object creation site



M

< graph$_{M1}$>

4 v5 = new <Application,LTestCases/SampleTests>@8(line 5)

N

< graph$_{N1}$>

The name of the allocation-site must be of the following form: **[<method-name>.new<instruction-number>]**

In graph$_{N1}$: **v5** → **[foo.new4]**

Eclipse

- You are encouraged to develop your tool within Eclipse.
- Within Eclipse, the arguments to PAVPointerAnalysis.java will be provided by the “Run Configuration” option in Eclipse (Run -> Run Configuration -> Arguments)

Relevant classes in WALA

- CallGraph
- CGNode
- IR
- SSAInstruction
- SSACFG
- ExplodedControlFlowGraph

Setting up the Project

- Follow the instructions in ProjectSetUp.pdf provided to you

Other Important Information

Other Information

- Do NOT import the com.ibm.wala.ipa.callgraph.propagation package.
  - Doing so will result in zero marks for the phase under evaluation, even if you do not use any class from the package
- Test Cases
  - A few provided right at the beginning
  - Evaluation to be done on more, previously undisclosed, test cases
- Do NOT modify any code region marked NO CHANGE REGION.
  - Example on next slide

Example

```
/*
* Skeleton main method. Initialize the variables appropriately and call the necessary functions.
* START: NO CHANGE REGION
*/
String classpath, mainClass, analysisClass, analysisMethod;

classpath = args[1];
mainClass = args[2];
analysisClass = args[3];
analysisMethod = args[4];

PAVPointerAnalysis pAnalysis = new PAVPointerAnalysis(classpath, mainClass, analysisClass, analysisMethod);
pAnalysis.runAnalysis();
// END: NO CHANGE REGION
```

Evaluation

- What we are looking for:
  - Your tool should not crash
  - The output should be sound
    - **Unsoundness**: *an edge OR node which should be in the points to set, is not present*
  - No unexpected imprecision
    - **Imprecision**: *an edge OR node which need not be in the points to set, is present*

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

Evaluation

- Demo of Phase 1:
  - Date: Will be announced soon.
  - During demo: *run your tool on predisclosed as well undisclosed (new) testcases*
- 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
- Copying will be dealt with severely
- Both teammates need to participate. During the demo, we will be evaluating the responses of both members.

References

- http://wala.sourceforge.net/javadocs/trunk/
- http://wala.sourceforge.net/wiki/index.php/User_Guide
- References on pointer analysis are given in the write-up

Acknowledgements

- We would like to thank Snigdha Athaiya and Suvam Mukherjee for allowing us to use their slides, and for providing the implementation of the initial set up of analysis and the instruction manual to set up WALA.

Thank you