# 03/05/21 Recitation Notes

17-355/17-665/17-819: Program Analysis (Spring 2021)
Jeremy Lacomis

## 1 Guaranteed Defs Analysis

For this recitation, we will be implementing a simple intraprocedural analysis that collects variables are guaranteed to be defined at each program point. This is similar to reaching definitions, but instead of caring about the location of a definition, we only care if a variable was defined at all. Our lattice and initial dataflow information is as follows:

$$\begin{array}{ccc} \sigma & \in & \mathcal{P}^{\mathsf{Vars}} \\ \sigma_1 \sqsubseteq \sigma_2 & \iff & \sigma_1 \subseteq \sigma_2 \\ \sigma_1 \sqcup \sigma_2 & = & \sigma_1 \cap \sigma_2 \\ \top & = & \mathsf{Vars} \\ \bot & = & \emptyset \\ \sigma_0 & = & \emptyset \end{array}$$

## 2 Soot/Jimple

For this recitation and the next homework we will be using Soot. Soot is a Java optimization and analysis framework for Java bytecode implemented in Java. It supports four different intermediate representations: Java Bytecode, BAF, JIMPLE, and GRIMP. You will analyze programs written in the JIMPLE IR, which is a typed 3-address code representation of Java bytecode.

### 2.1 Soot Example

The recitation repository contains a file `src/edu/cmu/se355/recitation/TestPrintDefs.java` that contains the following code:

```
public class TestPrintDefs {
    public void main(String[] args) {
        int x = 3;
        int y = x + 4;
        int z = (y - x) * 2;
        int w = 0;

        for (int i = 0; i < 10; i++) {
            w = w + 1;
        }

        if (x > 4) {
            y = 4;
        } else {
            x = 4;
        }

        System.out.println(x + y + z + w);
    }
}
```

After building the repository with `ant build`, we can run Soot and produce Jimple with the following command:

```
java -cp lib/soot-trunk.jar soot.Main -cp "build:lib/rt.jar" \
-f J edu.cmu.se355.recitation.TestPrintDefs
```

*(For Windows, you will need to replace* "build:lib/rt.jar" *with* "build;lib/rt.jar")

This will generate a Jimple file in the `sootOutput` directory:

```
public class edu.cmu.se355.recitation.TestPrintDefs extends java.lang.Object
{

    public void <init>()
    {
        edu.cmu.se355.recitation.TestPrintDefs r0;
        r0 := @this: edu.cmu.se355.recitation.TestPrintDefs;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return;
    }

    public void main(java.lang.String[])
    {
        edu.cmu.se355.recitation.TestPrintDefs r0;
        java.lang.String[] r1;
        int i0, $i1, $i2, $i3, $i4, i6, i7, i8;
        java.io.PrintStream $r2;
        byte b5;

        r0 := @this: edu.cmu.se355.recitation.TestPrintDefs;
        r1 := @parameter0: java.lang.String[];
        i6 = 3 + 4;
        $i1 = i6 - 3;
        i0 = $i1 * 2;
        i7 = 0;
        i8 = 0;

     label1:
        if i8 >= 10 goto label2;
        i7 = i7 + 1;
        i8 = i8 + 1;
        goto label1;

     label2:
        goto label3;

     label3:
        b5 = 4;
        $r2 = <java.lang.System: java.io.PrintStream out>;
        $i2 = b5 + i6;
        $i3 = $i2 + i0;
        $i4 = $i3 + i7;
        virtualinvoke $r2.<java.io.PrintStream: void println(int)>($i4);
        return;
    }
}
```

## 3 Implementing an Intra-procedural Dataflow Analysis in Soot

For getting started with Soot's dataflow framework, you can have a look at the GitHub wiki page, here (the majority of the information contained in these notes comes from this link, so refer to this link at your leisure):
https://github.com/Sable/soot/wiki/Implementing-an-intra-procedural-data-flow-analysis-in-Soot

### 3.1 The UnitGraph

Intra-procedural data-flow analyses operate on a control-flow graph for a single method, and in Soot this is called a UnitGraph.

A **UnitGraph** contains statements as nodes, and there is an edge between two nodes if control can flow from the statement represented by the source node to the statement represented by the target node.
A data-flow analysis associates two elements with each node in the UnitGraph, usually two so-called flow sets: one in-set and one out-set. These sets are:

1. Initialized
2. Propagated through the UnitGraph along statement nodes until,
3. A fix point is reached

**In the end, all you do is inspect the flow set before and after each statement**. By the design of the analysis, your flow sets should tell you exactly the information you need.

### 3.2 Types of Flow Analysis Implementations in Soot

There exist three different kind of FlowAnalysis implementations in Soot:

- ForwardFlowAnalysis – this analysis starts at the entry statement of a UnitGraph and propagates forward from there.
- BackwardsFlowAnalysis – this analysis starts at the exit node(s) of a UnitGraph and propagates back from there (as an alternative you can produce the InverseGraph of your UnitGraph and use a ForwardFlowAnalysis; it does not matter).
- ForwardBranchedFlowAnalysis – this is essentially a forward analysis but it allows you to propagate different flow sets into different branches. For instance, if you process a statement like `if (p!=null)` then you may propagate the into "p is not null" into the "then" branch and "p is null" into the "else" branch.

What direction you want to use depends entirely on your analysis problem. In the homework and in recitation we will use ForwardFlowAnalysis.

### 3.3 Implementing the Analysis Interface

To implement a forward flow analysis, we subclass ForwardFlowAnalysis.

### 3.3.1 Type Parameters

ForwardFlowAnalysis is a generic class with two type parameters:

`N`: The node type. Usually this will be `Unit`, but in general you are also allowed to have flow analyses over graphs that hold other kind of nodes.

`A`: The abstraction type ($\sigma$ type). Often times people use sets or maps as their abstraction ($\sigma$), but in general you can use anything you like.

*** **Beware though**, that your abstraction type must implement `equals(..)` and `hashCode()` correctly, so that Soot can determine when a fixed point has been reached!

### 3.3.2 Constructor

You have to implement a constructor that at least takes a `DirectedGraph` as an argument (where `N` is your node type; see above) and passes this on to the super constructor. Also, you should call `doAnalysis()` at the end of your constructor, which will actually execute the flow analysis. Between the super constructor call and the call to `doAnalysis()` you can set up your own analysis data structures.

```
GuaranteedDefsAnalysis(UnitGraph graph) {
    super(graph);
    // Initial empty GEN set
    unitToGenerateSet = new HashMap<Unit, FlowSet>(graph.size() * 2 + 1, 0.7f);

    // Iterate over the graph, computing gen sets for each Unit
    Iterator unitIt = graph.iterator();
    while (unitIt.hasNext()) {
        Unit s = (Unit) unitIt.next();
        FlowSet genSet = emptySet.clone();

        if (s instanceof DefinitionStmt) {
            Value leftOp = ((DefinitionStmt) s).getLeftOp();
            if(leftOp instanceof Local) {
                Local v = (Local) leftOp;
                genSet.add(v, genSet);
            }
        }
        unitToGenerateSet.put(s, genSet);
    }
    doAnalysis();
}
```

### 3.3.3 `newInitialFlow()` and `entryInitialFlow()`

The method `newInitialFlow()` must return an object of your abstraction type `A`. This object is assigned as the initial in-set and out-set for every statement, except the in-set of the first statement of your `UnitGraph`.

The in-set of the first statement is initialized via `entryInitialFlow()`.

```
/**
 * All INs are initialized to the empty set
 **/
protected Object newInitialFlow() {
    return emptySet.clone();
}

/**
 * IN (Start) is the empty set
 **/
protected Object entryInitialFlow() {
    return emptySet.clone();
}
```

### 3.3.4 copy(..)

The `copy(..)` method takes two arguments of type `A` (your abstraction), a source, and a target. It merely copies the source into the target. Note that the class `A` has to provide appropriate methods. In particular, `A` may not be immutable. You can work around this limitation by using a box or set type for `A`.

```
protected void copy(Object source, Object dest) {
    FlowSet sourceSet = (FlowSet) source;
    FlowSet destSet = (FlowSet) dest;

    sourceSet.copy(destSet);
}
```

### 3.3.5 merge(..)

The `merge(..)` method is used to merge flow sets at merge points of the control-flow (e.g., at the end of a branching statement such as if/then/else). It takes three arguments, an out-set from the left-hand branch, an out-set from the right-hand branch and another set, which is going to be the newly merged in-set for the next statement after the merge point. `merge(..)` acts like the join operator.

```
/**
 * Join == Intersection.
 **/
protected void merge(Object in1, Object in2, Object out) {
    FlowSet inSet1 = (FlowSet) in1;
    FlowSet inSet2 = (FlowSet) in2;
    FlowSet outSet = (FlowSet) out;

    inSet1.intersection(inSet2, outSet);
}
```

### 3.3.6 flowThrough(..)

The last method one needs to implement is the method `flowThrough(..)`. This method implements your actual flow function. It takes three elements as inputs:

- An in-set of type `A`.
- The node that is to be processed. This is of type `N` (usually a `Unit`).
- An out-set of type `A`.

The content of this method again depends entirely on your analysis and abstraction.

```
/**
 * OUT is IN plus the genSet
 **/
protected void flowThrough(Object inValue, Object unit, Object outValue) {
    FlowSet in = (FlowSet) inValue;
    FlowSet out = (FlowSet) outValue;
    in.union(unitToGenerateSet.get(unit), out);
}
```

*** **Note:** To see all flow-analysis implementations in Soot, browse all sub-classes of `FlowAnalysis`.

## 3.4 Helpful Soot Resources

- Soot Wiki
- The Soot Survivor's Guide
- Sable Thesis detailing Soot, includes a useful description of the JIMPLE intermediate representation
- Soot's Javadocs