02/08/19 Recitation Notes

17-355/17-665/17-819: Program Analysis (Spring 2019)

Jenna Wise
jlwise@andrew.cmu.edu

1 Reminders

- Homework 4 is due next **Thursday, February 14, 2019 at 11:59pm**. Instructions and the starter code can be found on the course website

2 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

2.1 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.

2.2 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.

2.3 Implementing the Analysis Interface

To implement a forward flow analysis, we subclass ForwardFlowAnalysis.

2.3.1 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!

2.3.2 Constructor

You have to implement a constructor that at least takes a DirectedGraph as an 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.

Listing 1: Reaching Defs Example

```
GuaranteedDefsAnalysis(UnitGraph graph)
    {
        super(graph);

        unitToGenerateSet = new HashMap<Unit, FlowSet>(graph.size() *
           2 + 1, 0.7f); // Initial empty GEN set
        unitToKillSet = new HashMap<Unit, FlowSet>(graph.size() * 2 +
           1, 0.7f); // Initial empty KILL set
        Map<Local, FlowSet> DEFS = new HashMap<Local, FlowSet>(); //
           Initial empty DEFS set

        // HINT/RECOMMENDATION:
```

```
        // It may be helpful to use full definition statements in
           place of x_m in the GEN, KILL, and DEFS sets

        // TODO: pre-compute the GEN sets for each program statement
        // TODO: pre-compute the DEFS set for pre-computing the KILL
           sets for each program statement
        // TODO: pre-compute the KILL sets for each program statement


        doAnalysis();
    }
```

2.3.3 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().

Listing 2: Reaching Defs Example

```
/**
 * All INs are initialized to ????
 **/
protected Object newInitialFlow()
{
    return // TODO: return All INs;
}

/**
 * IN(Start) is ????
 **/
protected Object entryInitialFlow()
{
    return // TODO: return IN(Start);
}
```

2.3.4 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.

Listing 3: Reaching Defs Example

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

```
            destSet = (FlowSet) dest;

        // TODO: implement data flow information copy; copying source
           into dest
    }
```

2.3.5 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 (if/then/else). Opposed to copy(..) 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.

Listing 4: Reaching Defs Example

```
/**
 * All paths joined by ????
 **/
protected void merge(Object in1, Object in2, Object out)
{
    FlowSet
        inSet1 = (FlowSet) in1,
        inSet2 = (FlowSet) in2,
        outSet = (FlowSet) out;

    // TODO: implement the join operator on data flow information
}
```

2.3.6 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, i.e. usually a Unit.
- An out-set of type A.

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

Listing 5: Reaching Defs Example

```
/**
 * OUT is ????
 **/
protected void flowThrough(Object inValue, Object unit, Object
   outValue)
{
    FlowSet
```

```
            in = (FlowSet) inValue,
            out = (FlowSet) outValue;

        // TODO: implement the flow function for reaching definitions
    }
```

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

2.4 Pair-programming Exercise

We have performed a pair-programming exercise in recitation. Here is some more information about pair-programming:

We often have 2 programmers to a single computer. One of the programmers is a "Driver", who controls the mouse/keyboard and deals with the details of the implementation. The other programmers is a "Navigator", who thinks at a higher level and watches for typos and/or logical errors. Switching between the Driver and Navigator positions happens roughly every 10-20 minutes.

Pair-programming is a good thing to do, because it creates better developers and better code. See this slide deck for more information on pair-programming.

2.5 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