02/22/19 Recitation Notes

17-355/17-665/17-819: Program Analysis (Spring 2019)
Jenna Wise
jlwise@andrew.cmu.edu

1 Reminders

- Homework 6 is due **Tuesday, March 5, 2019 at 11:59pm**. There is a checkpoint due next **Thursday, February 28, 2019 at 11:59pm**. Instructions can be found on the course website

2 Implementing an Interprocedural Analysis in Soot

2.1 Interprocedural Control Flow Graph in Soot

[1] We perform interprocedural analyses on *control flow graphs* that model the control flow of entire programs rather than single functions or methods.

These graphs contain additional edges for handling call and return instructions. For every call to function g, we add an edge from the call site to the first instruction of g, and from every return statement of g to the instruction following that call.

2.1.1 CallGraphs

Control flow graphs are referred to in Soot as **CallGraphs**. They can be accessed through the environment class (Scene) only in whole program mode. To make sure that your analysis is running in whole program mode use the **wjap** pack (you were using the **jap** pack in previous assignments):

```
PackManager.v().getPack("wjap").add(new Transform(ANALYSIS_NAME,
   SignAnalysis.instance()));
```

Once your analysis is set-up to run in whole program mode, it can access the application's (program's) CallGraph through the Scene via the getCallGraph() method:

```
Scene v = Scene.v();
CallGraph cg = v.getCallGraph();
```

The CallGraph class and other associated constructs are located in the **soot.jimple.toolkits.callgraph** package.

2.1.2 CallGraph Representation

A call graph in Soot is a collection of edges representing all known method invocations. Each edge in the call graph contains four elements:

[1] Some of the information and text in this section comes from the Soot Survivor's Guide (linked in Sec. 3 below), Section 7 on pages 30-33

1. Source method
2. Source statement (if applicable)
3. Target method
4. The kind of edge (the different kinds of edges are e.g. for static invocation, virtual invocation and interface invocation)

The call graph has methods to query for the edges coming into a method, edges coming out of method and edges coming from a particular statement:

- edgesInto(method)
- edgesOutOf(method)
- edgesOutOf(statement)

Each of these methods return an Iterator over Edge constructs. Soot provides three so-called adapters for iterating over specific parts of an edge:

- Sources iterates over source methods of edges
- Units iterates over source statements of edges
- Targets iterates over target methods of edges

For example, in order to iterate over all possible calling methods of a particular method, we could use the code:

```
public void printPossibleCallers(SootMethod target) {
    CallGraph cg = Scene.v().getCallGraph();
    Iterator sources = new Sources(cg.edgesInto(target));

    while (sources.hasNext()) {
        SootMethod src = (SootMethod)sources.next();
        System.out.println(target + " might be called by " + src);
    }
}
```

2.2 Context Sensitive Analysis for Recursive Procedures

We introduce context sensitivity to allow us to return analysis results to a caller that reflect the analysis results passed into the callee. We must also deal with the challenge of recursive functions or more generally, mutual recursion. The algorithm discussed in the rest of this section is a context sensitive analysis that handles mutual recursion/recursive functions (it is recommended that you implement a version of this algorithm for hw6).

2.2.1 Algorithm Overview

The algorithm overview is written in comments in the algorithm presentation below.

Listing 1: Context Sensitive Analysis for Recursive Procedures Algorithm

```
type Context
    val f_n : Function // function to be analyzed
    val input : L // input data flow information to the function f_n

type Summary // summary of the results from analyzing a context
    val input : L // input data flow information to the function f_n
                  // contained in a context we analyzed
    val output : L // output data flow information from the function f_n
                   // after analyzing the context it is in

val worklist : Set[Context]
val analyzing : Stack[Context] // keeps track of the contexts being
                               // analyzed; stack type due to
                               // nested function calls
val results : Map[Context, Summary] // data flow analysis results;
                                    // maps a context to the summary
                                    // of the results from analyzing it
val callers : Map[Context, Set[Context]] // maps a context to
                                         // its calling contexts

function AnalyzeProgram
    // initial context to be analyzed is the program entry method
       (main) with ⊤ input dataflow information
    worklist = { Context(main, ⊤) }

    // keep analyzing contexts in the worklist as long as it is not
       empty; removing them from the worklist before analyzing them
    while NotEmpty(worklist) do
        ctx = Remove(worklist)
        Analyze(ctx)
    end while
end function

// analyze the context, ctx, intraprocedurally
// ctx should be on the analyzing stack only while it is being analyzed
// if the new output data flow result for ctx after analyzing it is
   more general than the cached output data flow result for ctx, then
   the new output data flow result for ctx is cached in place of the
   currently cached result and the callers of ctx are added to the
   worklist
function Analyze(ctx, σ_i) // σ_i is the input dataflow info in ctx
    σ_o = results[ctx].output
    Push(analyzing, ctx)
```

```
    σ'_o = Intraprocedural(ctx)
    Pop(analyzing)
    if σ'_o ⋢ σ_o then
        results[ctx] = Summary(σ_i, σ_o ⊔ σ'_o)
        for c ∈ callers[ctx] do
            Add(worklist, c)
        end for
    end if
end function

// we need to define a precise flow function to handle method/function
   call instructions in our intraprocedural analysis
// this flow function computes the callee's context and then
   determines the data flow analysis result from analyzing the
   callee's context
// the data flow analysis result from analyzing the callee's context
   is used to compute the output information of the flow function,
   which is the input information to the flow function modified to map
   x to the data flow analysis result from analyzing the callee's
   context
// this flow function also adds ctx as a caller to the calleeCtx
function Flow([[n : x := f(y)]], ctx, σ_i) // σ_i here is the input info
                                           // to this flow function
    σ_in = [formal(f) ↦ σ_i(y)] // σ_in maps the formal parameters of f to
                                // their abstract values in σ_i
    calleeCtx = GetCtx(f, ctx, n, σ_in)
    σ_o = ResultsFor(calleeCtx, σ_in)
    Add(callers[calleeCtx], ctx)
    return σ_i[x ↦ σ_o[result]]
end function

// σ_i is the input data flow info for ctx
function ResultsFor(ctx, σ_i)
    σ = results[ctx].output
    // if there is cached output info for the current context, ctx,
       that isn't bottom and the cached input info for ctx is more
       general or equal to the argument (new) input info for ctx then
       the result of analyzing ctx is its cached output info
    if σ ≠ ⊥ ∧ σ_i ⊑ results[ctx].input then
        return σ
    end if
    // if the cached output info for ctx is bottom or ctx's new input
       info is more general than what is cached, then replace the
       cached input info for ctx with the argument (new) input info
       for ctx
    results[ctx].input = results[ctx].input ⊔ σ_i
    // if we are already analyzing ctx, then make the result of
       analyzing ctx again bottom; otherwise, analyze ctx and return
```

```
       the result (note that this is a recursive call to Analyze)
    if ctx ∈ analyzing then
        return ⊥ // has similar benefits to using ⊥ for initial
                 // data flow values on the back edges of loops
    else
        return Analyze(ctx)
    end if
end function

// return the context constructed for f based on f and σ_in, where σ_in
   is defined as above
function GetCtx(f, callingCtx, n, σ_in)
    return Context(f, σ_in)
end function
```

2.2.2 Algorithm Implementation Advice for Sign/Parity/Zero Analyses

***Note:** that this is just advice and you may choose to implement the algorithm differently, if you implement it at all

Listing 2: Data Type Advice

```
// Note: that I have chosen to not implement Context and GetCtx in
   such a way that limits context-sensitivity (see the lecture notes
   on Interprocedural Analysis for more information)
type Context
  val f_n: Function
  val input: L // L = Map<Local, LatticeType>
               // int locals/params to f_n mapped to correct abstract
                  value
type Summary
  val input: L // L = Map<Local, LatticeType>
               // int locals/params to f_n mapped to correct abstract
                  value
  val output: L // L = LatticeType
                // merged abstract values for int
                // f_n return values
                // after intraprocedural analysis of f_n

function GetCtx(f, callingCtx, n, σ_in)
  return Context(f, σ_in)
end function
```

Other Algorithm Implementation Advice:

- It is recommended to implement Context and Summary classes, which must have equals(Object obj) and hashcode() methods implemented

- It is recommended that method/function calls to which an ActiveBody is not available (outside the scope of the analysis) Top should be returned and the callee is not analyzed, ie. int x = Integer.parseInt("5")

3 Helpful Soot Resources

- Soot Wiki
  - Soot Wiki: Implementing an Intraprocedural Analysis in Soot
- The Soot Survivor's Guide
- Sable Thesis detailing Soot, includes a useful description of the JIMPLE intermediate representation
- Soot's Javadocs