Homework 3
Posted Friday February 14, Due Thursday February 27
50 points

Description of HW3 adds on immediately following the description of HW2.

1. Overview of Class Analysis Framework

Starter code in package analysis builds a Java class analysis framework on top of Soot. Code in Analysis.java abstracts away Jimple into 8 kinds of statements relevant to class analysis. Since we are interested in class analysis, i.e., flow of values of reference type, we ignore all statements and expressions on primitive types. All exposed variables (essentially all, there is one caveat) are of reference type.

1. Assignment: x = y
2. Field read: x = y.f
3. Field write: x.f = y
4. Array read: x = y[
5. Array write: x[] = y
6. Object allocation: x = new A
7. Direct call: either static invoke x = sm(args) or special invoke x = y.m(args)
8. Virtual call: x = y.m(args)

Analysis.java includes a worklist-like algorithm for solving constraints/transfer functions. Your task is to encode constraints as needed for a specific analysis then fire up the algorithm to compute the fixpoint solution. In the first assignment, HW2, you will implement Rapid Type Analysis, which entails the simplest constraints (i.e., transfer functions) and dataflow information (i.e., lattice). In HW3 you will implement XTA, a more precise and slightly more complex analysis. (If you need an extra challenge you can code 0-CFA or PTA, i.e., points-to analysis, as well.) In HW4 you will move on from toy programs to larger Java programs and compare RTA to XTA with respect to call graph construction, essentially trying to reproduce the results of Tip and Palsberg’s OOPSLA’00 paper: “Scalable Propagation-Based Call Graph Construction Algorithms”.

Folder programs contains 7 toy Java programs. Run the analysis on each of these and carefully examine the Jimples created by Soot. The autograder will test your analysis on these programs, as well as some other simple programs.

2. HW2

Your first task is to code Rapid Type Analysis which we covered in class. Place your code in package analysis.RTA. Starter code for this package is already there in your repo. Study the code, as well as the rest of the framework. Places where you will be adding your code are marked as TODO: YOUR CODE HERE. (There are other TODOs scattered throughout the code; these are reminders for me to fix, eventually.)

Once you are done, print the results of RTA on the console. Your analysis.RTA.showResults should print all RTA-reachable methods, by full name in alphabetical order, followed by all instantiated classes, also in alphabetical order. For example, the expected output from one of the toy programs looks like this:
Reachable methods:
=== <A: int add(A)>
=== <A: void <init>()>
=== <A: void m()>
=== <A: void main(java.lang.String[])>
=== <A: void sm()>
=== <B: void <init>()>

Instantiated classes:
=== A
=== B

When you are done, push into your repository and click Submit in Submitty. I will be running your RTAAnalysis with a slightly different driver to compare your output with the expected output.

3. HW3

Now that you are (more) familiar with Soot and the Class Analysis framework, you will build one of the more complex class analyses, XTA.

Add a new package analysis.XTA parallel to analysis.RTA. Add your XTA implementation in public class XTAAnalysis extends Analysis in this package. You may add analogous drivers to the ones in RTA to test locally. Submitty pulls your analysis/XTA/XTAAnalysis.java to test, so it is important to follow the required directory structure.

This is a rehash of the XTA constraints we discussed in class.

1. Allocation:
   1: for each new A in m s.t. m $\in$ ReachableMethods do
   2:   \{A\} $\subseteq S_m$
   3: end for

2. Virtual call:
   1: for each x = y.n(z) in m s.t. m $\in$ ReachableMethods do
   2:   for each C in SubTypes(StaticType(y)) $\cap$ S_m do
   3:     \{n\}'(this, p, ret) = resolve(C, n)
   4:     \{n\}' $\subseteq$ ReachableMethods
   5:     \{C\} $\subseteq S_{n'}$ // add receiver class to S_{n'}
   6:     S_m $\cap$ SubTypes(StaticType(p)) $\subseteq S_{n'}$ // add to S_{n'} due to parameter passing
   7:     S_m $\cap$ SubTypes(StaticType(ret)) $\subseteq S_m$ // add to S_m from S_{n'} due to return
   8: end for
   9: end for

I have simplified the presentation stating that a method has exactly one formal parameter p. Of course, a method may have 0 or more parameters and you do need to handle that. The class analysis framework passes all actual arguments of a method to the analysis (through the callbacks), including parameters of primitive type. You will need to do some extra work to filter out parameters of primitive type. Soot API methods getParameterType and getReturnType in SootMethod may be of help.
3. Field Read:
1: for each x = y.f in m s.t. m ∈ ReachableMethods do
2: \( S_f \subseteq S_m \)
3: end for

4. Field Write:
1: for each x.f = y in m s.t. m ∈ ReachableMethods do
2: \( S_m \cap \text{SubTypes}(\text{StaticType}(f)) \subseteq S_f \)
3: end for

In addition to these statements, you will need to handle direct call statements, static field reads and writes, and array reads and writes. Direct calls that are static calls are unambiguous. For direct calls that are instance calls (i.e., have a receiver), use the following constraint to pass the type of the receiver to the callee \( n \):
\[
S_m \cap \text{SubTypes}(\text{StaticType}(\text{this})) \subseteq S_n
\]

Soot API method \text{getDeclaringClass} may be of use here.

Static field reads, \text{local = static field}, and writes, \text{static field = local}, are abstracted away as \text{assignStmt} in the class analysis framework, where either the right-hand-side or left-hand-side node is of kind \text{STATIC FIELD}. You can either have separate constraints for static fields and instance fields (even though analysis of static and instance fields is exactly the same) or you can find a way to reuse field read/write constraints.

Finally, make sure you handle arrays as well; ignoring arrays would be unsound. Consider

```java
void m(X[] a) {
    X x = a[0];
    x.n();
}
```

and note that in general, the array argument may have been written \textit{anywhere} in the program.

Finally, as with RTA, display your result in \textit{showResult}. Specifically, display all reachable methods \( m \) in alphabetical order with all classes in \( S_m \) in alphabetical order. For example, the expected output for \textit{p2} is the following

Reachable methods:

\texttt{<A: int add(A)>}

\texttt{== A}

\texttt{== B}

\texttt{<A: void <init>()>}

\texttt{== A}

\texttt{== B}

\texttt{<A: void m()>}

\texttt{== A}

\texttt{== B}

\texttt{<A: void main(java.lang.String[])>}

\texttt{<A: void sm()>}

\texttt{== A}

\texttt{== B}

\texttt{<B: void <init>()>}
