Homework 3

Posted Monday February 11, Due Thursday February 21

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 a more complex class analysis, XTA.

Update your `hw02` repository. I have committed 3 more toy programs to test with; they should be showing in directory in `programs/p8`, `programs/p9` and `programs/p10`. Add a new package `analysis.XTA` and 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. Follow directory structure as Submitty pulls your `analysis/XTA/XTAAnalysis.java` to test.

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

1. Allocation:

for each `new A` in m s.t. $m \in \mathit{ReachableMethods}$ **do**
    $\{\mathtt{A}\} \subseteq S_m$
end for

2. Virtual call:

for each `x = y.n(z)` in m s.t. $m \in \mathit{ReachableMethods}$ **do**
    for each `C` in $S_m \cap \mathit{SubTypes}(\mathit{StaticType}(\mathsf{y}))$ **do**
        $n'(\mathsf{this}, \mathsf{p}, \mathsf{ret}) = \mathit{resolve}(\mathtt{C}, n)$
        $\{n'\} \subseteq \mathit{ReachableMethods}$
        $\{\mathtt{C}\} \subseteq S_{n'}$ // add receiver class to $S_{n'}$
        $S_m \cap \mathit{SubTypes}(\mathit{StaticType}(\mathsf{p})) \subseteq S_{n'}$ // add to $S_{n'}$ due to parameter passing
        $S_{n'} \cap \mathit{SubTypes}(\mathit{StaticType}(\mathsf{ret})) \subseteq S_m$ // add to S_m from $S_{n'}$ due to return
    end for
end for

I have simplified the presentation stating that each method has exactly one formal parameter `p`. Of course, a method may have 0 or more parameters and you do need to handle this. Note that the class analysis framework passes all actual arguments to the analysis, including ones that are of primitive type. You will have to do some extra work to filter out parameters of primitive type. Soot API methods `getParameterType` and `getReturnType` in `SootMethod` may be of use.

3. Field Read:

for each `x = y.f` in m s.t. $m \in \mathit{ReachableMethods}$ **do**
  $S_f \subseteq S_m$
end for

4. Field Write:

for each `x.f = y` in m s.t. $m \in \mathit{ReachableMethods}$ **do**
  $S_m \cap \mathit{SubTypes}(\mathit{StaticType}(\mathsf{f})) \subseteq S_f$
end for

In addition, you must handle direct calls, static fields and array reads/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:

$$S_m \cap \mathit{SubTypes}(\mathit{StaticType}(\mathsf{this})) \subseteq S_n$$

Static field reads, local = static_field, and writes, static_field = local, are abstracted as assignStmt in the class analysis framework, where either the right-hand-side or left-hand-side node is of kind STATIC_FIELD. This design choice may demand separate constraints for static fields even though handling of static and instance fields is exactly the same.

Finally, make sure you handle arrays! Ignoring arrays renders the XTA analysis unsound. Consider

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

and note that in general, the array argument may have been written *anywhere* in the program.

Finally, as with RTA, display your result in `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 `p2` is the following

```
Reachable methods:
<A: int add(A)>
=== A
=== B
<A: void <init>()>
=== A
=== B
<A: void m()>
=== A
=== B
<A: void main(java.lang.String[])>
<A: void sm()>
=== A
=== B
<B: void <init>()>
=== B
```