Homework 2

Posted Monday January 28, Due Monday February 11
50 points

1. Eclipse, Git, Soot and Starter Code

Download and install Eclipse Oxygen if you don't have it already. I will be using the same infrastructure for starter code and autograding as last semester's Principles of Software. I'm assuming you are familiar with Eclipse, Git and Submitty and will be including only minimal instructions.

Clone your hw02 Git repository in Eclipse:

(1) Go to File → Import.
(2) Select Git → Projects from Git.
(3) Select Clone URI and click Next.
(4) Enter URI: `https://submitty.cs.rpi.edu/git/s19/csci4450/hw02/YourRCSID`. YourRCSID is your RPI email id, e.g., milana5, NOT YOUR RIN.
(5) Enter your RCSID and password then click Next.
(6) In the Branch Selection dialog, click Next. (master should be selected)
(7) In the Local Destination dialog, select a target directory for the local Git repo (e.g., /Users/milanova/git/milana5_hw2), then click Next.
Important: Remember the target directory! You will need to type it in when creating the Eclipse Java project.
(8) In the dialog that appears, select "Import using the New Project wizard", then click Finish.
(9) In the New Project dialog, select Java → Java Project, then click Next. (This will start creation of a new Java project.)
(10) In Project Name, type "class-analysis", unselect "Use Default Location" then in Location, enter the target directory where the local Git repo resides (in our example, /Users/milanova/git/milana5_hw2), then click Finish. At this point, you should see a project named "class-analysis" [target_directory master] with subdirectory src. In directory src you should see packages `analysis` and `analysis.RTA`. These packages should remain under the Build path. However, subdirectory `programs` should not be under the build path. Select all `src/programs/pN`, then right click then Build Path → Remove.
(11) At this point, packages `analysis` and `analysis.RTA` show a bunch of errors because they are missing Soot libraries. Download the Soot jar from the course web page: `www.cs.rpi.edu/~milanova/csci4450/soot-develop.jar`. Then add it to the build path for your class analysis project: right click on class-analysis, then Build Path → Add External Archives... then select `soot-develop.jar`.
(12) Lastly, include JUnit to get rid of the JUnit errors: Right-click on project class-analysis, then Build Path → Add Libraries... → JUnit (JUnit 4) → Finish. At this point, you should be able to run that one JUnit test: right-click on RTATests.java, then select Run As → JUnit Test.

IMPORTANT: Make sure that your directory structure is correct. If not, compilation on Submitty will fail. You must have project `class-analysis` with subdirectory `src`. Directory `src`

must have subdirectory `analysis` and `programs`. In Project Explorer you should see packages `analysis` and `analysis.RTA` which are under the build path, and directory `programs` which contains the toy Java programs we'll analyze for a start. Directory `programs` should not be under the build path.

2. 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 (RTA), 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.

3. HW2

Your first task is to code Rapid Type Analysis (RTA), 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 analysis is done, print analysis results 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 `AnalysisRTA` with a slightly different driver to compare your output with the expected output.