Homework 4 (Programming): Dataflow Analysis

17-355/17-665/17-819O: Program Analysis
Claire Le Goues and Jonathan Aldrich
clegoues@cs.cmu.edu, aldrich@cs.cmu.edu

Due: Thursday, February 15, 2018 (11:59 PM) 100 points total

Assignment Objectives:

- Implement a dataflow analysis in a code framework based on the concepts of flow functions and lattices.

Handin Instructions. Place all your homework files in your private GitHub repository in a folder called `hw4`. Running `ant` in the `hw4` directory should automatically compile and test your analysis. After the deadline, we will clone your repository and run the `ant` script. You can test that you did everything correctly by running the following in a temporary directory:

```
git clone https://github.com/CMU-program-analysis/<YOUR-ANDREW-ID>
cd <YOUR-ANDREW-ID>/hw4
ant
```

1 Setup and Tool Information

Create a folder called `hw4` in your GitHub repository. Download hw4.zip from the course website and unzip it in the `hw4` directory. You'll be using Soot again, so make sure you have the same dependencies installed as those in hw1.

We have provided a test case, `TestSign.java`, which you can run with the `ant` command; passing the test case is an indication that you are on the right track, though earning credit for the assignment requires implementing your own analysis in a general way so that it will also work correctly with other test programs.

For getting started with Soot's dataflow framework, have a look at the Github wiki page, here:
`https://github.com/Sable/soot/wiki/Implementing-an-intra-procedural-data-flow-analysis-in-Soot`

2 Sign Analysis Implementation

In this assignment, you will implement your sign analysis for the Java programming language using Soot's dataflow analysis capabilities. You may choose to use some other dataflow analysis engine and/or some other language; if you want to do so, contact the instructor to discuss whether any aspects of the assignment need to be adapted.

In Java, integer variables are separate from variables that hold references, booleans, floating point values, etc. Your implementation need only track information for variables of type int.

Your analysis only needs to track the sign of local variables. Any use of fields, arrays, method parameters or results can be considered to have unknown sign.

You should implement your analysis by defining a class to represent your lattice. Your dataflow analysis will need to define the operations for your lattice, as well as the flow functions. Don't forget to ensure that your lattice handles equals() and hashCode() correctly!

In order to drive the flow analysis, use a BodyTransformer that creates a flow analysis for each method body. Inside the BodyTransformer, use tags to report the values of variables used as array indexes–e.g. whether they are definitely negative (an error) or possibly negative (conceptually a warning).

Additional notes on analysis requirements. Real-world languages typically include a variety of types that correspond to integer. You need only track information for variables that correspond to type `int` in Java or C.

Regardless of language, your analysis should cover variable copies, integer constants, addition, subtraction, multiplication, and division as precisely as possible given your lattice. Again, your analysis should reason about local variables; you may assume that globals or fields or similar, including the results of function calls or array accesses, and method parameters, are unknown. You are not required to correctly analyze other operations, though your analysis should not crash on code that includes them.

Question 1, Turn in your code, *(60 points).*

> Turn in your analysis code. Your code should follow the basic design described above. Use reasonable coding style and commenting such that we can figure out which part of your code does what and convince ourselves of its correctness. We reserve the right to run your analysis code on examples other than TestSign.java, looking both for accuracy of the analysis and its robustness (i.e. it should not throw unexpected exceptions when run on a larger codebase).

Question 2, Write test code, *(20 points).*

> Produce one or more standalone pieces of code in the language you are targeting to test your analysis (e.g., `MyTestSign.java`, `myTestSign.c`). You must include at least one method that includes a potential negative array index operation and at least one function with safe array accesses. You may include more than two functions (and should probably use more than two methods to test your code anyway); if your code enables me to find bugs in your fellow students' implementations, I will give you *extra credit*(1–10 points, depending on how tricksy it is). Submit the test code in a directory named `test` in the `hw4` folder (you can keep a copy of your test in the same directory as `TestSign.java` if it makes running tests easier for you). If your analysis framework requires that the target language be preprocessed, submit both the preprocessed and unpreprocessed versions of this code.

Question 3, Run your analysis on the test code, *(10 points).*

> Run your analysis on the test code you wrote for question 2. Capture the output of your analysis (as one or more text logs or screenshots, depending on how you wrote it) to

show the results of the analysis. If you take a screenshot, resize the window as necessary to show all the relevant output. Place screenshots (if any) in a `testOutput` folder in `hw4` and paste the output log in a file called `README.md`, also in the `testOutput` directory. Note you can embed screenshots (like .png) in the `README.md` on GitHub.

Question 4, Document how to setup and run your analysis, *(10 points).*

Include a `README.md` file in your `hw4` folder that explains how to set up your analysis and run it on a file, project, or similar written in the target language. Specify dependencies as necessary.