503: Program Analysis

Assignment 4: Abstract Interpretation Implementation

Due: **Monday**, May 8 before midnight

In the last assignment you designed an abstract interpretation to catch divide-by-zero errors. In this assignment you will implement your design (or a simplified version of it) for integer division in Java.

Implementation. You will build your implementation using the Checker Framework[1], a framework that simplifies the task of writing compiler plug-ins for Java.

1. Download the provided starter code at `https://courses.cs.washington.edu/courses/cse503/17sp/assignments/divide-by-zero-skeleton-code.zip`. The source code for the analysis is in a folder called "divbyzero." Run "`make test`" to compile the code and run the analysis on "programs/-Foo.java." You will need JDK 8 or higher. (The Makefile will automatically download a recent version of the Checker Framework, so no other installation steps are necessary.) You will not see any errors because the analysis is not finished yet.

2. Create the lattice structure. The lattice is defined declaratively using files in "divbyzero/qual," one file per point. Top has been given for you, but you will need to create the others yourself. (If you had designed an abstract interpretation with an infinite lattice, you will want to compress your design to a finite one.) The Checker Framework manual has more information on this if you get stuck.[2]

3. Implement error reporting. The file "DivByZeroVisitor.java" is responsible for reporting errors at specific places in the program once the analysis terminates.

4. At this point, your analysis is functional (although not very useful). Verify that errors are reported using "`make test`."

5. Implement the abstraction function. The file "DivByZeroAnnotatedTypeFactory.java" specifies the rules for what types to attach to constants in the program.

6. Implement refinement rules. The function `refineLhsOfComparison` in the file "DivByZeroTransfer.java" specifies the rules for how information is carried into if-statements. For example, the statement

   ```
   if (y ≠ 0) {
       System.out.println(x / y);
   }
   ```

   should not report any divide-by-zero errors.

7. Implement transfer functions for arithmetic. The function `arithmeticTransfer` in the file "DivByZeroTransfer.java" specifies the outputs for arithmetic expressions in terms of points in the lattice. For example, the statements

   ```
   int x = 1 + 0;
   int y = 1 / x;
   ```

   should not report any divide-by-zero errors.

[1] `https://checkerframework.org/`

[2] `https://checkerframework.org/manual/#creating-typequals`

Use and Writeup. To run your analysis, compile any Java program using the "javac" script provided by the Checker Framework that was downloaded by the Makefile, and ensure that it gets the flags `-classpath` *`path/to/skeleton-code`* and `-processor divbyzero.DivByZeroChecker`. The Makefile has an example of proper use that is printed when you run "`make test`." Find a Java project, either one of your own or open-source, with more than 1000 lines of code. Run your analysis on it to find potential divide-by-zero errors.

1. How many potential errors does your analysis report? Estimate how many are false positives.
2. Describe two different potential errors your analysis reported and explain why your tool reported errors at those locations.
3. Based on what you have found, suggest some potential improvements to your analysis that could make it more useful.

Submit your assignment to Canvas as a ZIP file containing your souce code and a PDF of your writeup. (You do not need to include the code you tested your analysis on, but you may provide a link if it is open-source.)

When you submit your completed assignment, please indicate how many hours it took to complete. This will not factor into your grade.

Useful Tips. When creating the lattice, you will need to attach annotations to each class:

- Every lattice point class should have `@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})`.
- The top of your lattice (and only the top!) should have `@DefaultQualifierInHierarchy`.
- Use `@SubtypeOf({Top.class, Other.class, ...})` to indicate *direct* subtype relationships. If you wouldn't draw those edges in the lattice diagram, you do not need to include them in the subtype list.
- The bottom of your lattice (and only the bottom!) should have the magical annotation

```
import org.checkerframework.framework.qual.ImplicitFor;
import org.checkerframework.framework.qual.LiteralKind;

@ImplicitFor(
    literals = {LiteralKind.NULL},
    typeNames = {java.lang.Void.class}
)
```

Functions you need to fill in throughout the code are at the tops of their respective files. Helper functions are available in some places, and code you do not need to touch is marked as "Checker Framework plumbing." However, you are welcome to extend your analysis in any way you choose, should you have the time and interest to do so.