Homework 5 (Programming): Context-Sensitive Interprocedural Analysis

17-355/17-665/17-819: Program Analysis
Fraser Brown, Ian Dardik

Checkpoint due: *Tuesday, October 08, 2024 (11:59 PM) [50 points]*
Final due: *Thursday, October 24, 2024 (11:59 PM) [200 points]*

Assignment Objectives:

- Implement a context-sensitive, interprocedural dataflow analysis.
- Handle the complexities of analyzing a real programming language.
- Make use of a real framework for analyzing Java code.

Handin Instructions. Submit your *entire* GITHUB repository for this assignment following the instructions on GradeScope. The assignment accepts GitHub repositories only, with a popup that links to your GitHub account and shows you a list of your available repositories. Note that this is different from the last coding assignment, where we asked you to submit a single file. This also means you must *commit and push* code and changes to GitHub to be able to submit the latest version for this assignment. Note that we do *not* plan to look *directly* at your GITHUB repository for this assignment, even though you do need to use it. You *must* submit your code via Gradescope. Gradescope does not automatically pull new versions of your code, so you must resubmit whenever you have a new version that you would like graded.

Your grade will be based on a combination of the autograder tests your code passes, and certain manual considerations, described at the end of this document.

1 Context-Sensitive Interprocedural Analysis Implementation

In this assignment, you will implement (and test!) a context-sensitive interprocedural integer sign analysis for Java, using the simpler/less precise domain we implemented in homework 3. Implement context-sensitivity using the *call string approach* with a maximum depth of 2.

We provide starter code for this assignment based on the Soot framework, which supports analysis and optimization of Java code and projects. We have provided a bit less scaffolding than we did last time, but there are still clear TODOs in comments indicating where you should begin your implementation. In particular, you need to implement:

- In `Context`, the **`getCtx`** method.
- In `Sigma`, the **`equals`** and **`hashCode`** methods(s).

- The majority of `IntraSignAnalysis`, which implements the intra-procedural part of the analysis. Note that you do *not* need to explicitly implement Kildall's, because Soot provides it at the backend; familiarize yourself with the framework, and look over the starter code to see what you *do* need to implement (e.g., flow functions, join, etc). *Unlike in homework 3*, you *do* need to implement a **`reportWarnings`** method, see below.
- The majority of `InterSignAnalysis`.

For the checkpoint, all you will need to implement is `Sigma` and `IntraSignAnalysis`

On language. In Java, integer variables are separate from variables that hold references, booleans, floating point values, etc. Your implementation need only track information for variables corresponding to type `int`, for local variables and method parameters. Your analysis should cover variable copies, integer constants, addition, subtraction, multiplication, and division as precisely as possible. You are not required to correctly analyze other operations, though your analysis should not crash on code that includes them. Your analysis should reason about local variables and method parameters; you may assume that globals, fields, or array accesses, are unknown.

Expected analysis output. At a high level, we expect the analysis to issue warnings when it identifies an array access that may involve a negative array index. This is the primary output we expect from your analysis implementation: a set of warnings for the code. We provide a standard method `Util.reportWarning` for reporting warnings, and there are example usages in `IntraSignAnalysis.java` that show how to call it. To see an example of how we will test this, you may look at the sample tests we provide in the starter code. The `src/test/inputs/` directory contains test inputs to test both intra- and interprocedural analyses; these also are commented where errors should be reported. The files `IntraAnalysisTest.java` and `InterAnalysisTest.java` show how we test the analysis results, and will be informative when you are writing your own.

Tests. We do provide sample tests. For the checkpoint, the tests provided in `IntraAnalysisTest.java` are the same as the autograder tests. For the final submission, we have a set of held-out autograder tests. For full credit on the final submission, you must *also* implement additional tests for your analysis, covering the key implementation considerations you are tackling. One or more test cases should require context sensitivity—i.e., the test case would fail if your analysis were interprocedural not context-sensitive. Your tests should use JUnit and be automatically run with `gradlew test`. We will download your Gradescope submission and run your tests.

2 Setup and Tool Information

Go to https://classroom.github.com/a/SxGwYgp9 to clone the starter code into your own private GitHub account. This assignment uses the Gradle build automation system. Gradle generates wrapper scripts that automatically download any dependencies that are needed to run a project, including Gradle itself.

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

Note that the Soot project was officially superceded by "SootUp", which was released in December 2022. However, the documentation is *extremely* sparse, and so we decided to stick with the deprecated project for this year. The Soot project has not yet been archived on GitHub and so you should be able to use it without problem in Fall, 2024.

Java version. Consult the README of the starter code for implications in terms of the versions of Java code that you can analyze with Soot, and be sure you're looking at the right documentation. A key problem is that Soot cannot analyze code past Java 15, but VMs for Java 12–16 are not maintained or have security vulnerabilities. This means that both codespaces and the autograder uses Java 11. Yes, it's old. Implication: avoid fancy language features from Java 14 on up.

2.1 Alternative 1 - Github Codespaces (Recommended)

Open this repository using GitHub Codespaces. To build the source code, open the Gradle Tab on the sidebar of Codespaces and clicking `build`.

We have provided test cases, which you can run with by clicking `test` on the same Gradle Tab. You can also set breakpoints before running the tests to debug your code. To enter debug mode, click on the debug icon in `test` option on the Gradle Tab.

2.2 Alternative 2 - Locally Building and Testing

Ensure you have installed the Java Development Kit. Anything from version 8 to 15 should work with Soot, but for consistency with the autograder, you probably want 11. To build the source code at the command line, you just need to run `./gradlew build` on *nix systems, or `gradlew.bat build` on Windows. If you use Visual Studio Code or IntelliJ, you will similarly need to configure it to use the correct version of Java (e.g., via modifying settings.json in Visual Studio code). You can then use the appropriate Gradle extensions to interact with the project.

We have provided test cases, which you can run with `gradlew test`; passing the test cases 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.

3 Grading

This assignment is worth 250 points in total. The checkpoint is worth 50 points, and the final is worth 200 points. Grading will involve both testing, and some manual assessment. The rough expected distribution of maximum available points for the automatically graded components is:

Checkpoint

- Correct implementation of **`Sigma`** and **`IntraSignAnalysis`**: 50 pts

Final

- Correct implementation of **`getCtx`**: 20 pts
- Correct implementation of **`Sigma`**: 10 pts
- Correct implementation of **`IntraSignAnalysis`**: 60 pts
- Correct implementation of **`InterSignAnalysis`**: 80 pts

The rough expected distribution of points for the manually graded components is:

- *New* tests that roughly cover the implemented functionality and run with `gradlew test` or with the `test` option on the Gradle Tab in codespaces: 20 pts

- Good coding practices, including code structure and commenting: 10 pts

Partial credit will be available as well, for all of the above.