Lab 2: Call-Graph Algorithms and Pointer Analysis

In this lab, we are venturing out from *intra*-procedural analysis towards *inter*-procedural analysis. In order to provide support for inter-procedural analyses, a common data structure – the call graph – must be constructed. A call graph is a static abstraction of all method calls that may happen at runtime. The nodes in such a graph are methods and the edges are potential calls. As the edges are recorded and stored in any order they do not represent the order in the program anymore. Therefore, a call graph is generally speaking flow-insensitive.

In this lab, we will investigate several call-graph algorithms that all solve the problem of polymorphic and indirect call sites, in general, with varying precision. The more imprecise a call graph is the more edges that cannot happen at runtime it includes. For example: a call graph contains edges to two possible methods at a polymorphic call site. The receiver object of the method call at this call site, however, can only be an instance of one specific class, then one edge in the call graph is correct and the other one is incorrect, therefore an imprecision (or overapproximation) in the call graph.

Of course, the Soot framework contains implementations for call graph algorithms. Please refrain from using these implementations in your implementations for this lab. The goal for this lab is that you implement them yourself. We have checks in place that tell us, that you do.

General Instructions
Set up: The code in folder `DECALab2-Soot` contains a maven project readily set up. To set up your coding environment:

- Make sure that you have any version of Java running on your machine.
- Import the maven project into your favorite development environment.
- To run the test cases, run the project as a Maven build, with the goals set to "clean test".
- **Important: Make sure that your test cases run via the command line.** To do so, run `mvn clean test` in the `DECALab2-Soot` directory.

Project:

The project contains:

- four `JUnit` test classes in the package `test.exercises` of the folder `src/test/java`. Your goal is to make all of those test cases pass. Do not change the test cases. Also, it is not advisable to implement a direct fit (e.g., a fixed call graph returning the expected outcomes instead of an algorithm to construct it) to the test cases, as we will grade your implementation.
- target classes in the packages `target.exercise*.` of the folder `src/test/java`. These files contain interesting cases for call graph algorithms.
- four classes that represent the algorithms you need to implement here. We added a few classes for data structures and common infrastructure. Do not change the data structure and infrastructure classes, only the algorithm classes. However, you are free to add new

classes, wherever you feel the need. All is located in the package `analysis` of the folder `src/main/java`.

Instructions

- Do **not** rely on the class names or methods from the test cases in your implementation. We have some more tests to check your implementations, which prevent you from getting the full amount of points if these fail too. In a nutshell: Just because all of the provided test cases are green, it does not mean that you are totally correct.
- Do **not** limit the entry points your analysis processes. Always process everything that the call to `this.getEntryPoints(scene)` provides you. Your analysis should be able to process any code correctly, not just the test cases.

Exercise 1

Implement the Class Hierarchy Analysis (CHA) algorithm into the `populateCallGraph` method in class `CHAAlgorithm`.

The CHA algorithm requires a class hierarchy to resolve polymorphic call sites. Start with constructing this data structure.

To identify method calls, have a look at the body of a method where `hasActiveBody()` returns `true`. You will see different types of invocations in the units that you may need to treat differently.

The latest documentation from Soot can also help: `https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/jdoc/`

(4 points)

Exercise 2

Implement the Rapid Type Analysis (RTA) algorithm into the `populateCallGraph` method in class `RTAAlgorithm`.

As RTA only resolves calls to classes that have actually been instantiated. Thus, you first need to determine which classes that are.

(2 points)

Exercise 3

Implement the Variable Type Analysis (VTA) algorithm into the `populateCallGraph` method in class `VTAAlgorithm`. Initialize the algorithm with the return from your CHA implementation. We have prepared this in the exercise frame.

Start with implementing your construction of the Type Assignment Graph. You will need that when resolving method calls. Be aware that you only need the graph for invocations that have a target, i.e. not for static method calls.

To focus your efforts on the VTA algorithm, we provided you with an implementation you can use for the Type Assignment Graph. However, we do not check for the use of this data structure and you are free to implement one yourself.

Some implementation advice: Apparently, the `equals/hashCode` implementation of the `FieldRef` type is surprising. You may need to compare their signatures instead. We considered this in method `TypeAssignmentGraph::createId` already, but you may need this at another point in your implementation, too.

(4 points)

Exercise 4

Implement a Spark-like algorithm that constructs points-to sets and takes them into consideration when building the call graph. Fill in your solution into the `populateCallGraph` method in class `SparkLikeAlgorithm`.

(0 points - This exercise will not be graded. But it is a good way to strengthen your insight.)