apl. Prof. Dr. Thomas Noll | Christina Jansen, Christoph Matheja

General Remarks

- Please hand in your solutions in groups of 3.
- **Please submit your solution (including code and all required outputs) via email to**

  `matheja@cs.rwth-aachen.de`

  with subject SPA exercise submission. Do not forget to provide the matriculation numbers of all group members.
- Code that does not compile or that contains modified method signatures will be graded with 0 points.
- If you have questions regarding the exercises/lectures, feel free to write us an email or visit us at our office.

The goal of this exercise is to implement your own version of the interval analysis from lecture 7 to analyze Java Bytecode programs. Since Java Bytecode consists of more than 200 statements, we will use an existing framework, called Soot[1], that already provides implementations of parsers, fixed point computations, etc. This exercise sheet provides the essential information needed to get started with Soot. Further details are found on the Soot webpage. In particular, we recommend to have a look at the tutorials[2] and the API documentation[3].

Exercise 1 (Getting Started with Soot): (1 Points)

a) The Soot framework as well as a small framework for implementing your interval analysis are available as a git repository. To obtain all necessary files, run

```
git clone https://anonymous@srv-i2.informatik.rwth-aachen.de/scm/git/spa1617
```

on your terminal (passwort: anonymous). The most important directories of the cloned project are the following:

- resources/ contains the soot library (`soot-trunk.jar`),
- src/ contains a small code skeleton for the interval analysis you have to implement,
- examples/ contains a small set of example files to test your implementation

After you obtained all necessary files, you should test whether Soot works correctly on your machine by running the following command from the root directory of the cloned project:

```
java -cp resources/soot-trunk.jar soot.Main
```

If no errors occur, you can move to the next tasks.

b) Soot takes a Java Bytecode program, i.e., a bunch of class files, and creates a corresponding program in a simplified version of Java Bytecode, called Jimple, that is designed to simplify static analysis. For example, Jimple requires no stack and groups together several similar Bytecode statements. To create a Jimple file, run (from within the `examples` directory):

```
java -cp ../resources/soot-trunk.jar soot.Main -cp .:PATH-TO-YOUR-rt.jar B -f J
```

where `PATH-TO-YOUR-rt.jar` denotes the path to your Java run-time. This should create a folder called `sootOutput` containing the Jimple code of class `B`. **Please submit an unmodified copy of the Jimple file you obtained for class B together with your solution**.

[1] `https://sable.github.io/soot/`
[2] `https://github.com/Sable/soot/wiki/Tutorials`
[3] `https://ssebuild.cased.de/nightly/soot/javadoc/`

c) Soot already contains several simple static program analyzes. For example, you can annotate the previously generated Jimple file with the results of a live variable analysis by running (from within the examples directory):

```
java -cp ../resources/soot-trunk.jar soot.Main -print-tags -cp .:PATH-TO-YOUR-rt.jar
                          -f J -p jap.lvtagger on B
```

Again, please submit an unmodified copy of the Jimple file you obtained for class B together with your solution.

Exercise 2 (Implementing an Interval Domain): (9 Points)

The remainder of this exercise is dedicated to implementing the interval analysis (without conditional branching) presented in lecture 7 into Soot. The essential classes are already provided in the source code of the cloned repository. Every method you have to implement within a class is marked with a `TODO` comment that further specifies the expected behavior of the respective method. Please do **not** change any existing method signatures and implement **all** methods marked with a `TODO` as specified such that your code can be properly evaluated. Of course you may add additional variables, methods, and classes in your implementation. In particular, make sure to properly implement the `toString()` methods.

a) Before we can implement the actual analysis, we have to implement the underlying domain presented in the lecture. Implement the class `Interval`. This class should support basic operations on intervals, such as addition, subtraction, etc (cf. lecture 7). For example, `Interval.plus(a,b)`, where `a.toString() = ”[-inf,5]”` and `b.toString() = ”[5,inf]”`, should yield an Interval object that is represented by the string ”`[-inf,inf]`”.

b) Implement the class `IntervalDomain` that is initialized with the set of all variables occurring in a program (of type `Value` due to Soot's terminology). `IntervalDomain` represents an element of the complete lattice used for interval analysis as presented in the lecture.

c) Implement the class `IntervalAnalysis` that contains the necessary operations to perform the analysis. This includes the following methods:

- `merge(IntervalDomain in1, IntervalDomain in2, IntervalDomain dest)`:
  Copy the result of merging two domain elements (`in1`, `in2`) into `dest`. You should implement two variants of this method (in the respective branches provided in the code): First, use the least upper bound between `in1` and `in2`. Second, use the widening operator instead.
- `flowThrough(IntervalDomain in, Unit unit, IntervalDomain out)`:
  Perform a single step of the abstract semantics on the domain element `in`. The statement to be executed is specified by `unit`. The result of this execution has to be copied into `out`. The only relevant statements for interval analysis are assignments that are represented by units of type `AssignStmt`. You can obtain the variable x and the expression a of an assignment $x = a$ by the methods `getLeftOp()` and `getRightOp()` of class `AssignStmt`. In order to evaluate the right-hand side of an assignment, you also have to implement an evaluation of *expressions*. You should at least support variables (type `Local`), constants (type `Immediate`) as well as common arithmetic expressions (`AddExpr`, `SubExpr`, `MulExpr`).
- `entryInitialFlow()`:
  Returns the initial element to start the analysis.
- `newInitialFlow()`:
  Returns the initial element of all labels prior to the analysis.

d) Run your analysis on all example files provided in the `examples/` directory with widening enabled[4]. If implemented correctly, you should get a Jimple file for every class in which all relevant statements are

[4] It suffices to run the class `Main` together with a command line argument containing the name of the class to analyze. The comments in class `Main` provide further details.

annotated with the results of your interval analysis. **Please submit these Jimple files together with your solution.**

Hint: You can run your analysis by adding the name of a `.class` file, e.g. A if the class file is named A.class, as a command line argument. Alternatively, if no command line arguments are provided, all examples in the examples/ directory will be analyzed.