Software Analysis

Summer Term 2020, Sheet 1

UNIVERSITÄT PASSAU
Prof. Dr. Gordon Fraser
Stephan Lukasczyk

Publication Date: 2020–05–04
Submission Deadline: 2020–05–29, 12:00:00 UTC+2 (extended!)
Task Introduction: 2020–05–06

Please read this information on the exercise class carefully! Exercise sheets are individual assignments. Working together is considered plagiarism, which automatically results in failing this course. We use automated tools to check for plagiarism. You have to submit your solution *before* the submission deadline mentioned on each exercise sheet. If you submit late, the submission of that particular sheet cannot be considered for grading. To successfully submit an assignment you have to provide the following:

1. Push all your code (and any further necessary files to build and run your implementation) to your GitHub repository of the assignment. You are responsible that all necessary files are in the repository. We consider the provided Docker image the reference platform, which will also be used for grading.
2. Send an email to stephan.lukasczyk@uni-passau.de that consists of:
   Your name and surname, your student ID number (Matrikelnummer), the number/name of the exercise task, and the link to your GitHub repository (GitHub user name is fine as well).
3. Fill out the submission form (provided on Stud.IP), sign it and send it electronically (as PDF) attached to your submission email.

A submission is only considered full and valid—and thus will be considered for grading—if all three items above are fulfilled successfully before the deadline.

If you have questions on the assignment please ask them *as soon as possible*. We prefer questions on Stud.IP or the exercise class because this will enable all students to participate in discussions and receive detailed explanations.

Task 1: Static Slicing Tool

In our first assignment we will implement the static slicing algorithm, scheduled for the lecture on May 6, for Java byte code. Program slicing computes a set of program statements, called *program slice*, that may affect the values at some point of interest, the *slicing criterion*. Slicing was originally introduced by Mark Weiser [Wei79; Wei81; Wei84]. Since then many different approaches have been introduced. A comprehensive overview over the various slicing techniques can be found in the survey of Silva [Sil12].

For this assignment we will implement an intra-procedural static slicer for Java byte code. The tool takes as input a program P—an arbitrary Java class file in an arbitrary package available in the class path—and a slicing criterion S. The slicing criterion S of a

program P is a tuple $\langle s, V \rangle$, where s is a program statement and V is a variable set of P. The output of the tool is the static backward slice for the variables in V from the method start to the program statement s.

In practice, various techniques exist to compute a slice. A popular—and straight forward—technique is to use a *Program Dependence Graph* (PDG). It combines control and data dependencies into one graph structure [FOW87]. To calculate a slice for a slicing criterion S on the program's PDG one has to only traverse backward on the PDG starting at the node of the PDG that corresponds to the criterion's location.

1 Provided Framework

We provide a basic framework for this assignment via GitHub Classroom:

https://classroom.github.com/a/zOCXPmYU

Use this as a basis for your implementation. The base repository consists of a Gradle project with some basic implementation of input parsing and output generation.

1.1 Input/Output Handling

We provide you all input and output handling for the tool as well as a predefined (and fixed) interface the input/output handling relies on. The tool will be implemented as a command-line application. It will take the arguments presented in Table 1; note that for simplicity the implementation shall only consider one variable instead of a set in the slicing criterion.

The tool's output is determined by your implementation. The given output routine expects a set of nodes, returned by method `public Set<Node> backwardSlice(final Node pNode)` of the `ProgramDependenceGraph` class. We provide you the extractors that take this set of nodes—not necessarily ordered—and generates the expected output representation from it.

1.2 Libraries

The Gradle project provides the following libraries as dependencies: for byte code handling the ASM [BLC02; Kul07] library, Google's Guava library[1], an ASM extension for data-flow analyses[2], the Apache Commons CLI library[3], the JGraphT library[4], as well as JUnit 5[5], Google Truth[6], Hamcrest[7], Mockito[8], and PowerMock[9] for testing. The usage of further libraries is *not* permitted!

[1] https://github.com/google/guava
[2] https://github.com/saeg/asm-defuse
[3] https://commons.apache.org/proper/commons-cli/
[4] https://jgrapht.org
[5] https://junit.org
[6] https://github.com/google/truth
[7] http://hamcrest.org/JavaHamcrest/
[8] https://site.mockito.org
[9] https://powermock.github.io

Table 1: Arguments of the slicing tool

Argument	Description
`-c`, `--class`	Path to the class file, either as `org.example.Foo` or `org/example/Foo` for a class `Foo` in package `org.example`. The class/package has to be available in the Java class path.
`-v`, `--variablename`	Name of the variable in the slicing criterion
`-m`, `--method`	Method name and descriptor of the method we want to slice from. Expected syntax: `<methodname>:<descriptor>`, e.g., `foo:(I)I`
`-l`, `--linenumber`	Line number of the statement in the slicing criterion
`-s`, `--sourcefile`	*(optional)* Path to the source file of the class we apply our slicing to. This parameter is optional. If it is given, the tool's result will be the lines of the source code that are part of the slice; otherwise, the result will be presented in terms of the byte code.
`-t`, `--targetfile`	*(optional)* Path to a target file where the tool shall write the result. This parameter is optional. If it's not given, the tool writes to standard output.

1.3 Examples

The provided project bundles a selection of example tasks. You can find the input files in package `de.uni_passau.fim.se2.examples`. In the folder `expected-results` in the project's root you can find for each of the example tasks a text file that states the input and expected output of the tool.

Note: A minimal requirement regarding your implementation's functionality is to achieve the same results. A submission that cannot solve these examples will be graded zero points for functionality. Hard-coded values result in zero points for the whole assignment!

2 Your Task

This section shall cover the information necessary for you to implement the tool.

2.1 Implementation

Your task is to implement the static slicing algorithm presented in the lecture. This means, slicing is performed by traversing backward on the PDG starting at a given location, the slicing criterion.

In order to achieve this you have to implement the following functionality—all of which will be covered in the lecture on May 6—into the given framework:

- Calculate the static backward slice, that is, a set of nodes from the PDG beginning at a given node. This is done in the method `Set<Node> backwardSlice(final Node pNode)` of the `ProgramDependenceGraph` class. The parameter of the method is the node representing the slicing criterion. It returns the set of nodes that are the static backward slice. The output routine does not expect a specific ordering of the elements in this set. The node `pNode` is part of the backward slice.
  In order to calculate the slice you need a program-dependence graph.
- Implement the calculation of the program-dependence graph in the class `ProgramDependenceGraph` as shown in the lecture. For this, you need the data dependences. We provide you the class `DataFlowAnalysis` that implements this, hence it is not required you implement a data-dependence analysis.
  The program-dependence graph is a combination of these data dependences and the control dependences. Thus you need the control dependences.
- Implement the calculation of the control-dependence graph in the class `ControlDependenceGraph` as shown in the lecture. For this you need a post-dominator tree.
- Implement the calculation of the post-dominator tree in the class `PostDominatorTree` as shown in the lecture.

All three classes, the `PostDominatorTree`, the `ControlDependenceGraph`, and the `ProgramDependenceGraph` extend the abstract class `Analysis` which sets up the *control-flow graph* (CFG) in its constructor and provides the abstract method `computeResult()` as its public interface. It is not required to change the `Analysis` class. At all locations you have to provide an implementation you will find to-do comments as well as `RuntimeExceptions` that inform you of a missing implementation.

The framework provides you an implementation of a `Node` type for nodes, the type `ProgramGraph`, which basically wraps a graph in the JGraphT library, and a `CFGExtractor` that creates the CFG from the Java byte code. Furthermore, the framework provides you some additional helpers related to the CFG and to finding the right node for the slicing criterion. It is not permitted to change these classes, neither is it permitted to change the parameter parsing nor the output routines.

2.2 Tests

You are required to provide unit tests for all code you implement. You are *not* required to provide unit tests for code we provide to you. This means, you have to implement unit tests for the classes `PostDominatorTree`, `ControlDependenceGraph`, and `ProgramDependenceGraph`. If you add additional helper classes you have to implement unit tests from them as well. You are required to use the JUnit 5 framework for testing. Furthermore, you can use Google Truth or Hamcrest for more readable assertions and Mockito or PowerMock for mocking. It is not required to provide tests for the implementation we provide to you, since you are not allowed to change this implementation. Particularly, you are neither allowed to change the following classes and packages nor do you have to provide tests from them: `DataFlowAnalysis`, all classes in the

`de.uni_passau.fim.se2.slicer.util.cfg` and `de.uni_passau.fim.se2.slicer.util.output` packages, and `SlicerMain`. Furthermore, you do not need to provide tests for the example programs in the package `de.uni_passau.fim.se2.examples`.

The goal of testing is to provide regression tests that achieve full line and branch coverage on all your code. To evaluate coverage we will use JACOCO. The GRADLE task `gradle test` is configured in a way that it will generate the JACOCO HTML report automatically in `build/jacoco/html`. Please put your tests in `src/test/java` using an appropriate packaging structure.

Note: The provided tests bundle a test class called `ExpectedResultsTest`. This class is a system test that executes the public examples and compares them to their expected outputs. It will fail as long as you have not implemented the necessary functinality.

2.3 Docker Container and Evaluation Scripts

In order to provide you a fixed platform for your testing, which will also be the same as used during the grading process, we provide a `Dockerfile` to you, from which you can build a Docker image. Executing this image will automatically execute the evaluation script on the public functional test cases from the `examples` package. The script provides to you the achieved coverage as well as whether or not your implementation passes the individual functional tests. You can build the image with the command `docker build -t slicer .` and execute it with `docker run slicer`. A necessary requirement to use this is having the Docker software installed.

3 Submission and Evaluation

3.1 Your Submission

For a full submission that will be graded you are required to provide all necessary code—and, if necessary, all further required files—in your GitHub repository. Afterwards, send an email containing all information described in the preamble of this sheet to me.

All steps are necessary for a full submission that will be considered for grading. Missing one item results in not getting a grade for this sheet. *Note:* in order to receive a grade for the course you are required to submit solutions to all four assignments! This assignment is one of these four graded assignments.

3.2 Evaluation

During evaluation we will check the functional properties of your implementation, i.e., if it provides the same results as our reference implementation. A minimum requirement to get points for functionality is that your tool can solve all provided examples correctly. Hard-coded solutions will result in zero points for functionality. Besides the provided examples we will use further, secret, test cases to check your tool's functionality.

Furthermore, we incorporate the coverage achieved by the unit tests you provide into the evaluation results. We will use branch and statement coverage as provided by JACOCO.

Note that we will use automated plagiarism checkers. If you commit plagiarism you will fail the course.

References

[BLC02] E. Bruneton, R. Lenglet and T. Coupaye. 'ASM: A Code Manipulation Tool to Implement Adaptable Systems'. In: *Adaptable and Extensible Component Systems*. 2002.

[FOW87] J. Ferrante, K. J. Ottenstein and J. D. Warren. 'The Program Dependence Graph and Its Use in Optimization'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (1987), pp. 319–349. DOI: 10.1145/24039.24041.

[Kul07] E. Kuleshov. *Using the ASM Framework to Implement Common Java Bytecode Transformation Patterns*. 2007.

[Sil12] J. Silva. 'A Vocabulary of Program Slicing-Based Techniques'. In: *ACM Computing Surveys* 44.3 (2012), 12:1–12:41. DOI: 10.1145/2187671.2187674.

[Wei81] M. Weiser. 'Program Slicing'. In: *Proceedings of the 5th International Conference on Software Engineering (ICSE '81, San Diego, CA, USA, March 9–12)*. IEEE Computer Society, 1981, pp. 439–449.

[Wei84] M. Weiser. 'Program Slicing'. In: *IEEE Transactions on Software Engineering* 10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[Wei79] M. D. Weiser. 'Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method'. AAI8007856. PhD thesis. Ann Arbor, MI, USA, 1979.