Deep Static Modeling of `invokedynamic` (Artifact)

George Fourtounis
University of Athens, Department of Informatics and Telecommunications, Greece
`http://gfour.github.io/`
gfour@di.uoa.gr

Yannis Smaragdakis
University of Athens, Department of Informatics and Telecommunications, Greece
`http://yanniss.github.io/`
smaragd@di.uoa.gr

Abstract

Java 7 introduced programmable dynamic linking in the form of the `invokedynamic` framework. Static analysis of code containing programmable dynamic linking has often been cited as a significant source of unsoundness in the analysis of Java programs. For example, Java lambdas, introduced in Java 8, are a very popular feature, which is, however, resistant to static analysis, since it mixes `invokedynamic` with dynamic code generation. These techniques invalidate static analysis assumptions: programmable linking breaks reasoning about method resolution while dynamically generated code is, by definition, not available statically. In this paper, we show that a static analysis can predictively model uses of `invokedynamic` while also cooperating with extra rules to handle the runtime code generation of lambdas. Our approach plugs into an existing static analysis and helps eliminate all unsoundness in the handling of lambdas (including associated features such as method references) and generic `invokedynamic` uses. We evaluate our technique on a benchmark suite of our own and on third-party benchmarks, uncovering all code previously unreachable due to unsoundness, highly efficiently.

2012 ACM Subject Classification Software and its engineering → Compilers; Theory of computation → Program analysis; Software and its engineering → General programming languages

Keywords and phrases invokedynamic, lambdas, static analysis

Digital Object Identifier 10.4230/DARTS.5.2.6

Funding We gratefully acknowledge funding by the European Research Council, grant 790340 (project PARSE).

Related Article George Fourtounis and Yannis Smaragdakis, "Deep Static Modeling of invokedynamic", in 33rd European Conference on Object-Oriented Programming (ECOOP 2019), LIPIcs, Vol. 134, pp. 15:1–15:28, 2019.
`https://dx.doi.org/10.4230/LIPIcs.ECOOP.2019.15`

Related Conference 33rd European Conference on Object-Oriented Programming (ECOOP 2019), July 15–19, 2019, London, United Kingdom

1 Scope

This artifact contains the evaluation benchmarks for the paper "Deep Static Modeling of `invokedynamic`" by G. Fourtounis and Y. Smaragdakis (ECOOP '19). The benchmarks duplicate the results of the evaluation section of the paper (instructions follow in Section 1.1 and Section 1.2).

Installation is covered in Section 2.1. Solutions to common errors can be found in Appendix A.

The full sources of Doop (including the Datalog logic for the analysis) are included in the directory contained in the DOOP_HOME environment variable. The sources of the benchmarks are also included. All sources can be modified and rebuilt (consult Section 1.1.2 and Section 1.2.2 for information on how to inspect or extend the artifact).

1.1 Analyzing the Microbenchmark Suite

To analyze the microbenchmark suite (Section 5.1 in the paper), run:

```
cd ${HOME}/invokedynamic-benchmarks
./analyze-microbenchmarks.sh
```

1.1.1 Expected output

The programs should be analyzed without errors and a table with times should be printed (same as Figure 8 in the paper). A list of successfull tests should also appear ("Running check: ..."). The analysis of these programs covers the set of features mentioned in Section 5.1 of the paper. The output also mentions the features checked successfully ("Feature: ...").

1.1.2 Inspection and extension

For further manual inspection or to extend the artifact:

- The test programs are integrated as tests run by Doop's Gradle build system. Thus, these tests pass when all analyses have completed successfully and all feature checks have also completed successfully.
- The sources of the test programs are under ${DOOP_HOME}/tests.
- The test scenarios for the features are found in the following locations:
  - ${DOOP_HOME}/src/test/groovy/org/clyze/doop/TestLambdasMethodReferences.groovy
  - ${DOOP_HOME}/src/test/groovy/org/clyze/doop/TestInvokedynamic.groovy
  - ${DOOP_HOME}/souffle-logic/addons/testing/TestInvokedynamic.dl
- The Jimple intermediate representation can be found in ${DOOP_HOME}/out/context-insensitive/test-X/facts/jimple, where X is one of "104-method-references", "107-lambdas", and "115-invokedynamic". This is only available when `-generate-jimple` is given in the tests (instead of `-Xlow-mem`).
- To inspect more information computed by the analysis, declare the appropriate relation(s) in ${DOOP_HOME}/souffle-logic/main/export.dl and repeat the analysis. The information will be written to ${DOOP_HOME}/out/context-insensitive/test-X/database (see previous note for the values of X). The feature checks also validate results read from this directory.

1.2 Analyzing the Dynamic Test Suite

To analyze the test suite of Sui et al. (Section 5.2 in the paper), run:

```
cd ${HOME}/invokedynamic-benchmarks
./bench-invokedynamic.sh
```

1.2.1 Expected output

The programs should be analyzed without errors. The script generates the table of Figure 9 in the paper.

1.2.2 Inspection and extension

For further manual inspection or to extend the artifact:

- The benchmark sources can be found in directory ${HOME}/invokedynamic-benchmarks/dynamic-benchmark.
- The Jimple intermediate representation can be found in ${DOOP_HOME}/out/context-insensitive/X/facts/jimple, where X is one of "lambda-consumer", "lambda-function", "lambda-supplier", and "dynamo-reflection". This is only available when `-generate-jimple` is given in bench-invokedynamic.sh (instead of `-Xlow-mem`).
- To inspect more information computed by the analysis, declare the appropriate relation(s) in ${DOOP_HOME}/souffle-logic/main/export.dl and repeat the analysis. The information will be written to ${DOOP_HOME}/out/context-insensitive/X/database (see previous note for the values of X).
- The expected output is hard coded in file "bench-invokedynamic.sh" and the metrics (that should match this output) are calculated by the logic in file "dynamic-benchmark.dl".

2 Content

The artifact package includes file "invokedynamic-benchmarks.ova", which is a VirtualBox image ("appliance") that can be directly imported by the VirtualBox software.[1]

2.1 Installation

- Import "invokedynamic-benchmarks.ova" in VirtualBox via "File" / "Import Appliance".
- Start the virtual machine.
- When the virtual machine starts and the desktop appears, open a terminal (menu / "System Tools" / "LXTerminal"). The virtual machine contains a "user" account (password: "user") with sudo capabilities, so that additional software may be installed.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `https://gfour.github.io/invokedynamic-artifact/`.

4 Tested platforms

This artifact is bundled as a VirtualBox image (tested with VirtualBox 5.2.18) and should thus work on any platform supported by VirtualBox.

Script execution times vary significantly depending on the hardware used. We give here the times for two extremes, a slow laptop (system A) running the scripts natively and a fast server (system B) running the scripts inside the VirtualBox image.

System	OS	CPU	RAM
A	Ubuntu 16.04	Intel Core2 Duo CPU T6570, 2.10GHz	8G
B	Ubuntu 18.04	Intel Xeon Gold 6136 CPU, 3.00GHz	629G

[1] `https://www.virtualbox.org/`

The total execution times of the scripts follow in the table below:

System	analyze-microbenchmarks.sh	bench-invokedynamic.sh
A	31min 27sec	38min 39sec
B	17min 47sec	19min 25sec

5 License

The artifact is available under The Universal Permissive License (UPL), Version 1.0, Copyright (c) 2017 PLAST lab, University of Athens and Martin Bravenboer, except for file `MethodReferences Test.java` (Copyright (c) 2013, Oracle and/or its affiliates[2]).

6 MD5 sum of the artifact

fc01866198a461b2743a6aad4a2607f1

7 Size of the artifact

2.3 GiB

A Solutions to common errors

- The virtual machine may run out of disk space if benchmarks are run manually and all analysis results are kept. In that case, delete ${DOOP_HOME}/out or ${DOOP_HOME}/cache to make space and rerun the analysis.
- If the microbenchmark suite fails with a Gradle lock error, there may be a stale Gradle process still running. Stop it ("kill -9 <PID>") and rerun the analysis.

[2] `https://docs.oracle.com/javase/tutorial/displayCode.html?code=https://docs.oracle.com/javase/tutorial/java/javaOO/examples/MethodReferencesTest.java`