How functional is your program?

António Menezes Leitão

March, 2019

1 Introduction

Despite the success of object-oriented programming, we now witness a resurgence of functional programming. This happens, in large part, due to the bugs inherent to the imperative nature of most object-oriented languages, including Java and C++. In these languages, objects tend to be mutable entities and the state of the program is encoded in the fields of those objects. The state changes when assignments are done to those fields but, unfortunately, it is difficult to reason about the program's state when it can be changed from any point of the program.

In functional programming, however, assignments are forbidden and, thus, it is easier to ensure that a program is correct. In fact, given two different implementations of an application, the one written in a functional style is probably the one with fewer bugs. Moreover, functional programming can be easily combined with object-oriented programming, as languages like Clojure and Scala have demonstrated. This might give the impression that, in order to use functional programming, one has to switch to one of those languages but, in fact, that is not the case. What is important is to adopt the functional programming paradigm even if you are forced to work with a language that was originally designed with a different approach in mind. The most recent versions of Java also prove this point, as the language's designers recognized the importance of functional programming and decided that Java must support several functional programming features, including lambdas and streams. By using these features, we can significantly reduce the number of places where state-changes occur and, thus, avoid bugs. Unfortunately, it is almost impossible to completely avoid assignments in Java, as object creation, in general, requires that the object initialization must be done using assignments to its fields. Thus, to be fair, these assignments should not be considered a bad practice, since Java does not support any alternative to initialize an object.

The goal of this project is to develop a profiler that evaluates a Java application and computes metrics that give a rough idea of the amount of functional programming that was used in its development. To that end, the Java application is instrumented at load-time to count all operations that read and write fields of the application's objects. These operations should not include assignments to an object that are done only in its construction, i.e., that are visible in any of the object's constructors. The application is then executed and, at the end, it prints statistics regarding those read and write operations. The fraction between write operations and read operations gives an approximate answer to the question: is this application written in an imperative style or is it written in a functional style? When the fraction is 0 (no writes) the program is highly functional. When the fraction is 1 (one write for every read) the program is highly imperative.

As an example, consider the following program:

```
interface Counter {
    public int value();
    public Counter advance();
}

class ImperativeCounter implements Counter {
    int i;

    ImperativeCounter(int start) {
        i = start;
    }
    public int value() {
        return i;
    }
    public Counter advance() {
        i = i+1;
        return this;
    }
}

class FunctionalCounter implements Counter {
    int i;

    public FunctionalCounter(int start) {
        i = start;
    }
    public int value() {
        return i;
    }
    public Counter advance() {
        return new FunctionalCounter(i+1);
    }
}

public class Example {
    public static void test(Counter c1, Counter c2) {
        System.out.println(String.format("%s %s", c1.value(), c2.value()));
    }
    public static void main(String[] args) {
        Counter fc = new FunctionalCounter(0);
        test(fc, fc.advance());
        Counter ic = new ImperativeCounter(0);
        test(ic, ic.advance());
    }
}
```

The previous program demonstrates the different semantics produced by counter when a functional implementation is used or when an imperative implementation is used. When executed, the program prints:

```
0 1
1 1
```

However, when executed under the control of the profiler, it also prints:

```
Total reads: 6 Total writes: 1
class FunctionalCounter -> reads: 3 writes: 0
class ImperativeCounter -> reads: 3 writes: 1
```

Notice that the profiler prints not only the total of reads and writes that were done, but also the corresponding numbers for each class, sorted by the fully qualified name of the class.

2 Goals

The main goal of this project is the implementation of the profiler in Java, using Javassist. The classes, interfaces, and annotations should be implemented in the package `ist.meic.pa.FunctionalProfiler`.

You must also implement a Java class named `ist.meic.pa.FunctionalProfiler.WithFunctionalProfiler` containing a static method `main` that accepts, as arguments, the name of another Java program (i.e., a Java class that also contains a static method `main`) and the arguments that should be provided to that program. The class should (1) operate the necessary transformations to the loaded Java classes so that when the classes are executed they also count the number of reads and writes to the fields of the class' objects, and (2) should transfer the control to the `main` method of the program.

Remember that the field writes that are visible in a constructor and that serve to initialize a created object are **not** counted.

2.1 Extensions

You can extend your project to further increase your grade above 20. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Examples of interesting extensions include:

- Limiting the scope of the instrumentation via annotations
- Provide more detailed statistics regarding which fields are more heavily read or written
- Provide more detailed statistics regarding fields that are read or written outside of the methods of their classes

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections. In order to ensure this behavior, you should implement all your extensions in a different package named `ist.meic.pa.FunctionalProfilerExtended`.

3 Code

Your implementation must work in Java 8.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

4 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 10-minute slot, should be centered in the architectural decisions taken and may include all the details that you consider relevant. You should be able to “sell” your solution to your colleagues and teachers. You should be prepared to answer questions regarding both the presentation and the actual solution. If necessary, you will be asked to discuss your work in full detail in a separate session.

5 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named as `project.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing:

- The source code, within subdirectory `/src/main/java`
- A Gradle file `build.gradle` that, upon execution of the task build, generates a `functionalProfiler.jar` in the `build/libs/` folder.

Note that it should be enough to execute

```
$ gradle compileJava build
```

to generate (`functionalProfiler.jar`). In particular, note that the submitted project must be able to be compiled when unziped.

The only accepted format for the presentation slides is PDF. This PDF file must be submitted using the Fénix Portal separately from the ZIP file and should be named `p1.pdf`.

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner workings of the developed project, including demonstrations.

The public presentation of the project is a compulsory evaluation moment. Absent students during project presentation will be graded zero in the entire project.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code must be submitted via Fénix, no later than 19:00 of **March**, **29**. Similarly, the presentation must be submitted via Fénix, no later than 19:00 of **March**, **29**.

The presentations will be done during the classes after the deadline. Only one element of the group will present the work. The element will be chosen by the teacher just before the presentation. Note that the grade assigned to the presentation affects the entire group and not only the person that will be presenting. Note also that content is more important than form. Finally, note that the teacher may question any member of the group before, during, and after the presentation.