IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs

Thomas Durieux, Martin Monperrus

IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs

Thomas Durieux, Martin Monperrus

February 11, 2016

To refer to this document: Thomas Durieux and Martin Monperrus. "IntroClassJava: A Benchmark of 297 Small Java Programs for Automatic Repair", Technical Report #hal-01272126, University of Lille. 2016.

Abstract

Reproducible and comparative research requires well-designed and publicly available benchmarks. We present IntroClassJava, a benchmark of 297 small Java programs, specified by Junit test cases, and usable by any fault localization or repair system for Java. The dataset is based on the IntroClass benchmark and is publicly available on Github.

1 Introduction

Context Reproducible and comparative research requires well-designed and publicly available benchmarks. In the context of research in fault localization and automatic repair, this means benchmarks of buggy programs [5], where each buggy program comes with a specification of the bug, usually as test cases. Le Goues and colleagues [3] have published IntroClass a benchmark of small C programs written by students at the University of California Davis. The programs are 5-20 lines, usually in a single method, and are specified by a set of input-output pairs.

Objective We aim at providing a Java version of IntroClass, called IntroClassJava. IntroClassJava would allow to run and compare repair systems built for Java, such as Nopol [2].

Method We use source to source transformation for building IntroClassJava. There are two main transformations: the first one transforms the program itself, and esp. takes care of correctly handling pointers and input parameters. The second one transforms the input-output specification as standard JUnit test cases. We discard the programs that cannot be transformed. The resulting programs are standard Maven projects.

Result We obtain a benchmark of 297 Java programs usable by any fault localization or repair systems for Java. The dataset is publicly available on Github [1].

2 Methodology

We present how we build the IntroClassJava benchmark.

2.1 Overview

We take the IntroClass benchmark. We remove the duplicate programs, we transform the C programs into Java using an ad hoc transformation. We check that each resulting Java program has the same behavior as the original C program, by executing them using the inputs provided with IntroClass. Also, we transform the input output test cases of the C program into standard JUnit test cases.

When the transformed program does not compile, or has an observable behavior that is different from the original one (according to the provided input-output specification), we discard it. That is the reason for which IntroClassJava contains less programs than IntroClass.

2.2 Translating Pointers in Java

Many programs of IntroClass use pointer manipulation on primitive types, such as `*i` or `&i` if i is an integer. We use the following transformation to emulate this behavior in Java. The key idea of the transformation is to encapsulate every primitive value into a class. For instance, for integer this results in:

```
class IntObj {
    int value;IntObj
    IntObj (int i) {value = i;}
    ...
}
```

Then all variable declarations, assignments and expressions are transformed as follows.

C code	Java code
int i = 0	IntObj i = IntObj(0)
k = i;	k.value = i.value
int* j = &i;	IntObj j = i

We use this schema to handle C types int, float, long, double and char (using respectively `IntObj`, `FloatObj`, `LongObj`, `DoublObj`, and `CharObj`).

Listing 1: Handling of standard input and output

```
class Median {
  Scanner scanner; // for handling inputs both in test and command line
  String output = ""; // for handling the output

  static void main (String[] args) throws Exception {
      Median mainClass = new Median();
      String output;
      if (args.length > 0) {
          mainClass.scanner = new java.util.Scanner (args[0]);
      } else {
          // standard input
          mainClass.scanner = new java.util.Scanner (System.in);
      }
      mainClass.exec ();
      System.out.println (mainClass.output);
  }
  void exec () throws Exception {
    // the program code
  }
}
```

2.3 Testability

In IntroClass, the program are tested using standard command line arguments. For instance, to test a program computing the median of three numbers, one calls `$ median 4 0 17`. However, we aim at achieving standard Java testability using JUnit test cases. Consequently, we transforms the programs and test cases as follows.

2.3.1 Backward-compatible Main Method

We aim at supporting both command line arguments and JUnit invocations. This is achieved as follows. The main method of the Java programs takes either one argument or no argument. In the former case, it is meant to be used in a test case, in the latter case, it reads the arguments from the standard input as a Java program. We introduce an "exec" method for each program that is responsible of doing the actual computation. The "exec" method is used in test cases directly and in the main method when invoked in command line. Listing 2 shows an example main method.

2.3.2 Test Cases

In IntroClass, the specification of the expected behavior is based on input-output pairs, where each input and each output is in a separate file. For instance, file `1.in` contains the input of the first input/output pair and fail `1.out` the expected output.

We write a test case generator that reads those files and create Junit test classes accordingly. Consequently, there is one Junit test method per input-output pair. Each test case ends with the assertion corresponding to checking the actual output against the expected one. An example test case is shown in

Listing 2: Example of generated test case

```
class WhiteBoxTest {
  @Test (timeout = 1000)
  // corresponds to file "whitebox/1.in" of IntroClass
  public void test1() throws Exception {
    Smallest mainClass = new Smallest();
    String expected =
      "Please enter 4 numbers separated by spaces > 0 is the smallest";
    mainClass.scanner = new java.util.Scanner ("0 0 0 0");
    mainClass.exec ();
    String out = mainClass.output.replace ("\n", " ").trim ();
    assertEquals (expected.replace (" ", ""), out.replace (" ", ""));
  }
  //.. other test
}
```

Listing 2.

Note that the original output specification of IntroClass contains the command line invitations, they are kept as is in the generated test cases.

IntroClass contains two kinds of tests: blackbox and whitebox. Blackbox are manually written and whitebox ones are generated ones with symbolic execution with the golden implementation as oracle. Consequently, IntroClassJava contains two test classes: WhiteBoxTest and BlackBoxTest.

2.4 Unique Naming

In order to be able to analyze and repair all programs of the benchmark in the same virtual machine, we give each program and each test class an unique name based on the original identifier of IntroClass. It follows the convention subject, user, revision. For instance, class `smallest_5a568359_004` is a program for problem "smallest", written by user 5a568359, for which it was the fourth revision. All programs are in package "introclassJava".

2.5 Implementation

The transformations are implemented in Python. To analyze C code, we transform it into XML suing srcML [4]. srcML encodes the abstract syntax trees of programs as XML. We then load the resulting XML using a DOM library and generate the Java file using standard string manipulation. The printf formatting instructions are transformed to their Java counterpart which are mostly compatible (e.g. "String.format ("%d is the smallest", a.value)"). All programs of IntroClassJava are regular Maven projects.

3 Results

Final benchmark After applying the transformation described in Section 2, we obtain a dataset of 297 Java programs, all specified with JUnit test cases, with at least one failing test cases. Table 1 gives the main statistics of this dataset.

Group	# of buggy programs
checksum	11 programs
digits	75 programs
grade	89 programs
median	57 programs
smallest	52 programs
syllables	13 programs
Total	297 programs
only black box failing tests	33 programs
only white box failing tests	39 programs
both white box and black box	225 programs
Total	297 programs

Table 1: Main descriptive statistics of the IntroClassJava benchmark

Project	# wb ok	# wb ko	# bb ok	# bb ko	# both ko
checksum	0	11	4	7	7
digits	15	60	24	51	36
grade	1	88	0	89	88
median	9	48	6	51	42
smallest	7	45	5	47	40
syllables	1	12	0	13	12
Total	33	264	39	258	225

Table 2: Full breakdown by project

It first indicates the breakdown according to the subject program. For instance, in “checksum”, there are 11 programs that are failing. It then indicates the breakdown according to the failure type. For instance, there are 33 programs that which only white tests are failing. Table 2 gives the full breakdown by project where ”wb” means whitebox, ”bb” means blackbox, ”ko” means failing and ”ok” means passing.

Comparison with IntroClass In IntroClass, there are 450 failing programs, it means that the automatic transformation has failed for $450 - 297 = 153$ programs. Either the resulting program was not syntactically correct Java and consequently does not compile, or the transformation has changed the execution behavior. Interestingly, for some programs, the transformation itself repaired the bug (the C program fails on some inputs, but the Java version is correct). This is mostly due to the different behavior of the non-initialized variable in C versus Java. In C, the non-initialized local variables have the value of the allocated memory space and in Java, all primitive types have a default value (e.g. 0 for numbers).

4 Conclusion

We have presented how we have set up IntroClassJava, a benchmark of 297 buggy Java programs. It results from the automated transformation from C to Java of the IntroClass benchmark. This dataset can be used for future research on fault localization and automatic repair on Java programs. The benchmark is publicly available on Github [1].

References

[1] The github repository of introclassjava. `https://github.com/Spirals-Team/IntroClassJava`, 2015.

[2] F. DeMarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014)*, 2014.

[3] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering (TSE), in press*, 2015.

[4] J. I. Maletic, M. L. Collard, and A. Marcus. Source code files as structured documents. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC)*, 2002.

[5] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the International Conference on Software Engineering*, pages 234–242, 2014.