Formal Autograding in a Classroom

Dragana Milovančević, Mario Bucev, Marcin Wojnarowski, Samuel Chassot, and Viktor Kunčak

EPFL, Station 14, CH-1015 Lausanne, Switzerland

Abstract. We report our experience in enhancing automated grading in an undergraduate programming course using formal verification. In our experiment, we deploy a program verifier to check the equivalence between student submissions and reference solutions, alongside the existing, testing-based grading infrastructure. We were able to use program equivalence to differentiate student submissions according to their high-level program structure, in particular their recursion pattern, even when their input-output behaviour is identical. Consequently, we achieve (1) higher confidence in correctness of idiomatic solutions but also (2) more thorough assessment of solution landscape that reveals solutions beyond those envisioned by instructors.

Keywords: Functional programming · Automated grading · Program equivalence.

1 Introduction

With the growing numbers of students in programming courses, automated grading of programming assignments has become ubiquitous. The goal of practical and rigorous automated grading has motivated the development of tools to aid teaching programming courses [56,54,49,35,62,1,24]. These tools commonly employ automated testing to assess the correctness of student submissions [38].

We observed this general trend in the functional programming course at our university. Whereas seven years ago the course counted 200 students, this number has more than doubled since then. To maintain a reasonable workload for the teaching staff, the course heavily relies on testing to automate the assessment of programming assignments, both in the final exam and throughout the semester.

While highly automated, testing-based grading comes at the cost of accuracy and feedback quality. Researchers have shown that flaws in test suites and the consequential misclassifications of student solutions can have a negative impact on students [63]. Furthermore, among the solutions that do satisfy the input-output requirements, testing-based graders fail to provide solution-specific feedback [9,19]. Finally, in large programming courses, the use of automated grading tools can result in completely impersonal feedback, to the point where most students never get to speak with the instructors [22]. We would like to reverse this trend.

Like human graders, we want an automated grader that examines the source code instead of just running it. Ideally, we would like our automated grader to

help scaling human grading to support the growing number of students, while still providing meaningful, targeted feedback.

Providing feedback solely based on input-output tests is particularly problematic for solutions that are unusual, whether in a good or in a bad way. As an example, consider the student submissions from Figure 1, computing `gcd` for two natural numbers. The submission from Figure 1c exhibits the same input-output behaviour as all the other submissions from Figure 1, despite being less efficient. This submission would thus typically go unnoticed with a testing-based grader, despite its unique underlying approach. Furthermore, while all the submissions from Figure 1 have identical input-output behaviour, we can identify two different recursion structures (subtraction-based vs. modulo-based Euclid's algorithm). We would like our autograder to classify submissions not just by correctness, but also based on the underlying program structure.

Our main inspiration is the Rainfall study from functional programming education [15], which studies the variety in high-level structures of student submissions. The analysis from that study is a manual, time intensive process: it consists of first identifying categories of interest and then labelling each individual submission by hand. Whereas human insight is indispensable, this paper explores the potential for automation in producing such classification.

To this end, we suggest the use of formal verification tools for automated grading. We consider functional induction as the strategy of choice for proving program correctness, and recursion as the main indicator of the underlying program structure. We use the Stainless formal verifier for Scala, which was previously evaluated on recursive equivalence checking benchmarks, including programming assignments in an offline setting, translated from OCaml to Scala [39]. The theory and encouraging results of past programming languages research already suggest that verification tools can be used as grading assistants for program clustering [39,9], but the authors in past work do not go as far as deploying such tools in a classroom. In this paper, we report on our experiment using the Stainless verifier as a grading assistant in the *live setting* of an undergraduate programming course in our university. To encourage other researchers and educators to adopt formal verification tools in programming classrooms, we document our experience, what did, and what did not work in practice.

The main contributions of this paper are as follows:

- We report on our pilot experiment deploying the Stainless verifier as an automated grader for introductory exercises in a programming course. We describe our deployment process in detail and share our insights.
- We show that formal verification can reveal variations in the underlying program structure used in student solutions, such as different recursion schemas, even when solutions have identical input-output behaviour.
- We publish a new data set with over 700 Scala programs, alongside the exercise material.

```
def gcdR(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if a == b then a
  else if a > b then
    if b == 0 then a
    else gcdR(a − b, b)
  else
    if a == 0 then b
    else gcdR(a, b − a)
```

```
def gcdS(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if b == 0 then
    a
  else if a >= b then
    gcdS(a−b, b)
  else
    gcdS(b, a)
```

(a) Two programs from the cluster of submissions using subtraction-based Euclid's algorithm. Submission gcdS is proven equivalent to the reference solution gcdR, and therefore correct.

```
def gcdW(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if b == 0 then a else gcdW(b, a%b)
```

```
def gcdX(a: Int, b: Int): Int = b match
  case 0 => a
  case _ => gcdX(b, a % b)
```

```
def gcdY(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  var x = a
  var y = b
  while (y != 0)
    val temp = y
    y = x % y
    x = temp
  x
```

```
def gcdZ(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if a < b then
    gcdZ(b, a)
  if b == 0 then a
  else
    val r = a % b
    if r == 0 then b
    else gcdZ(b, r)
```

(b) Four programs from the cluster of submissions using modulo-based Euclid's algorithm, proven equivalent among themselves, and passing all the tests, but not proven equivalent to our reference solution gcdR. The instructor can manually inspect one submission from the cluster to provide feedback for the entire cluster (27 submissions).

```
def gcd(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  def checkGcd(a: Int, b: Int, testVal: Int): Int =
    require(testVal > 0)
    if a % testVal == 0 && b % testVal == 0 then testVal
    else checkGcd(a, b, testVal − 1)
  (a,b) match
    case (0,x) => x
    case (x,0) => x
    case (x,y) => if x < y then checkGcd(x,y,x) else checkGcd(x,y,y)
```

(c) Singleton cluster and suboptimal implementation. This submission passes the tests, but is not proven equivalent to any other submission. By failing to classify this program, the grader points to this submission that requires manual inspection, allowing the instructor to detect suboptimal implementation.

Fig. 1: Three clusters of student submissions for the gcd exercise, illustrating different recursion schemas.

```
def drop(ls: List[Int], n: Int): List[Int] =
  require(n >= 0)
  val helper = ls.foldLeft((Nil, 1))((base, elem) =>
    val list = base._1
    val count = base._2
    val newList = if count % 3 != 0 then list :+ elem else list
    (newList, count + 1)
  )
  helper._1

Unfortunately, we found some incorrect functions. Invalid functions:
  drop
    Counter−example with the following values:
    ls = (1, 2, 3, 4, 5, 6, 7, 8, 9, Nil), n = 0
    expected (1, 2, 3, 4, 5, 6, 7, 8, 9, Nil) but got (1, 2, 4, 5, 7, 8, Nil)
```

Fig. 2: An incorrect submission from the drop exercise. Stainless detects a concrete counterexample, which we communicate as feedback to the student, and use on the instructor side by adding it to the test suite.

2 Background: Stainless Verifier

One differentiating characteristic of our study is the use of a formal verifier. Our course uses Scala [45], so we adopt the Stainless verifier [34]. Stainless can prove program termination and the absence of runtime errors such as division by zero. Developers can optionally provide program specification using contracts, such as pre- and post-conditions. Stainless supports only a subset of Scala, with best support for a purely functional subset with ML-style polymorphism.[1] This subset largely aligns with the concepts that we present in our course.

Stainless works by first parsing and type checking the input program using the Scala compiler to generate an abstract syntax tree (AST). Stainless then transforms this AST into an equivalent purely functional program, from which it generates verification conditions using a type checking algorithm [27]. It iteratively unfolds recursive functions [60,57] and uses SMT solvers (Z3 [11], cvc5 [2], Princess [51]) to prove or disprove those verification conditions.

Our deployment makes use of a high-level functionality of Stainless to perform program equivalence checking via automated proofs by functional induction [39]. In this mode, rather than writing pre- and post-conditions, users provide specification in the form of a reference program. Stainless then attempts to prove program correctness via automated equivalence checks against the reference program. Our deployment also exploits the ability of Stainless to generate counterexamples for incorrect programs [60], providing feedback valuable to students and instructors (Figure 2).

[1] Stainless also supports imperative programs [26,5,52].

3 The Experiment

In this section, we describe our experiment deploying a Stainless-backed formal autograder in a second-year undergraduate course that teaches software construction through functional programming to around 400 students.

3.1 Teaching Software Construction Using Scala

The goal of our Software Construction course is to teach functional programming and reasoning about programs, along with software engineering concepts and skills. This includes concepts such as subtyping, polymorphism, structural induction, (tail) recursion, as well as soft skills like debugging, reading and writing specifications, or using libraries. During the semester, students work on 12 homework assignments, designed to produce interesting or practical programs, including games as web applications, dynamical system simulations, and file system traversals. In addition to these graded projects, students work on exercise sets, consisting of smaller problems designed to help grasp the course material, such as evaluation of algebraic expressions, manual recursion elimination, and memoisation.

3.2 Experimental Setup

Our experiment took place during the fall semester of 2023. We prepared four autograded exercises in the form of optional individual short programming assignments to be solved and submitted on a computer. Table 1 describes our exercises. We deployed the exercises in the last week of the course, one month before the final exam. This decision was made in part to avoid interfering with other aspects of the course, which was given for the first time in this form. We provided a dedicated section of the course's forum for questions and discussions about these optional exercises.

Students were invited to participate in the study by submitting their solutions, with an option to permit the public release of those submissions. The code was automatically graded by running tests and formal equivalence checks against our reference solution. We initially deployed one reference solution per exercise. During the experiment, we added another reference solution for the `drop` exercise, as discussed in Section 4. For each submission, students received automated feedback generated by our system. They were permitted to re-submit their solutions any number of times.

At the end of the course, we further examined the submissions using Stainless as a grading assistant on the instructor side. We gathered all the submissions proven correct, together with the reference solutions, and all the submissions that passed the tests, but the equivalence proof timed out against the reference solution. We then run the equivalence checking for each pair of programs in the entire set, to identify and analyse clusters of provably equivalent submissions.

Section 3.3 describes our deployment and Section 3.4 summarizes the results.

Table 1: Four exercises deployed in our experiment. #P: number of submissions per exercise. #F and LOC: average number of functions and lines of code per submission, respectively.

Name	Description	#P	#F	LOC
drop	Drop every n-th element from a list	373	1.8	13
gcd	Find the greatest common divisor	80	1.3	11
prime	Check if an integer is prime	220	3.7	19
infix	Implement infix boolean operators	46	7.8	17

3.3 Setting Up Stainless as a Grader

We next describe how we adapted the Stainless verifier to be used as a scalable on-premise grading service.

Grading Infrastructure. Our students use the Moodle educational platform to obtain and submit graded assignments. An internal Kubernetes service evaluates the submissions by running specified tests and uploading the grade and the feedback to Moodle. Naturally, we integrate our optional autograded exercises as Moodle assignments. This setup yields a simple process for the students, as there is no need to learn and use additional platforms. We use a custom Moodle plugin to automatically run Stainless as a service upon each assignment submission and to report grades and feedback to students. We create a dedicated Docker image for each exercise, packaged with Stainless and Z3 solver. An orchestrating script collects submissions, feeds them into Stainless along with the reference solutions, and produces feedback from the output of Stainless.

Exercise Setup. To illustrate the usability of our approach, we detail the steps needed to prepare each exercise:

1. Write the problem statement as a markdown file.
2. Set up a dedicated Scala project with the source files.
3. Write the reference solution(s).
4. Write MUnit tests for students to run their solutions locally.
5. Write a configuration file listing the name of the function(s) and the reference solution(s) for the equivalence checking.
6. Compile everything into a Docker image linked to Moodle.

Here, only Step 5 is specific to formal equivalence checks. The other steps were already in place for the existing testing-based grader.

Feedback Generation. For each submission, students receive a numerical grade, a feedback file with comments for each function, and a log file disclosing a detailed output of Stainless. The feedback file either contains a congratulation message, in case the submission is proven correct, or otherwise an explanation of the encountered error(s), such as in our example shown in Figure 2. We report custom feedback for the following errors, allowing students to iteratively improve their code until they solve each problem:

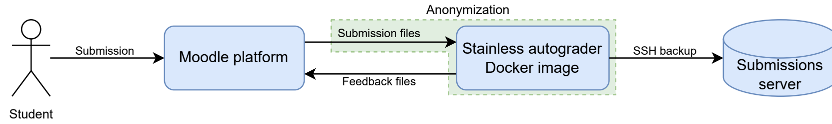


Fig. 3: Data collection pipeline.

- **User errors** – such as an incorrectly named file or an incorrect function signature
- **Safety errors** – such as division by zero or integer overflow
- **Counterexample errors** – a counterexample input was found, proving that the submission is incorrect
- **Termination errors** – a function could not be proven terminating within the specified timeout for SMT queries
- **Equivalence errors** – a function could not be proven equivalent to any reference solution within the specified timeout for SMT queries

Data Collection. Figure 3 shows our data collection pipeline. Upon each submission, the data is directly anonymized and copied out. Each submission is initially identified by a UUID, also included on Moodle to create a one-way link, allowing us to remove collected samples in case a student opts-out of participation.

3.4 Results

Table 1 shows the total number of submissions per exercise. Our data set comprising 709 Scala programs is available as supplementary material of this anonymous submission, and will be made publicly available under a permissive licence. We hope that the data set will be useful to evaluate future research on programming education, which lacks public data sets.[2]

We export the generated feedback and present the results per exercise in Table 2. Out of the 400 students taking the course, 201 students agreed to participate in the study. Several students actively engaged in discussions on the course forum, with over 70 posts total. While solving the exercises, some students submitted many attempts, and some students skipped some exercises, resulting in a total of 719 submissions. After removing byte-identical files, we were left with 709 submissions. Around one third of the submissions are not of interest due to compilation errors (Column S), resulting in 480 syntactically valid programs.

[2] In a recent systematic review [38], the authors analyse 121 research papers in the field and remark that, indeed, only 10 of them have publicly available data sets. We are grateful to our university's ethics committee and colleagues for their help throughout this study and to our students for allowing us to share their submissions. The overall process involved a significant administrative overhead, which may partly explain the scarcity of public data sets in the literature on programming education.

Table 2: Results of our experiment. S indicates submissions that could not be processed due to compilation errors, or due to students submitting wrong files. TS indicates submissions where Stainless could not prove safety checks. I and C show numbers of submissions proven incorrect and correct, respectively. TO indicates submissions where the equivalence check times out, left for clustering analysis. The last two columns show numbers of non-singleton and singleton clusters.

Name	S	TS	I	C	TO	Non-Singleton	Singleton
drop	60	56	169	14	70	10	26
gcd	21	2	8	3	42	2	12
prime	146	9	40	1	22	4	11
infix	2	0	2	42	0	1	0

Column I shows the number of incorrect submissions, evidenced by a counterexample (46%). Column C shows the number of submissions that our verifier proved equivalent to the reference solution, and therefore correct (13%). The remaining submissions are neither provably correct nor provably incorrect (42%), and are left for further manual inspection and clustering analysis.

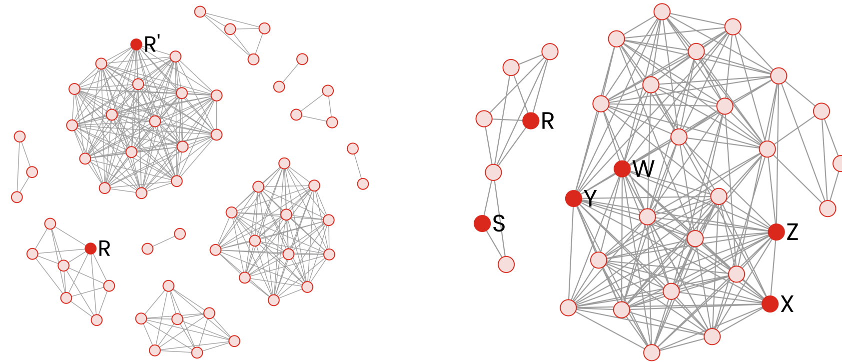
The last two columns show the results of our clustering analysis. Column "Non-Singleton" refers to the number of clusters with two or more equivalent submissions. Column "Singleton" denotes the number of submissions that are not provably equivalent to any other submission, and thus form a singleton cluster of their own.

4 Discussion and Lessons Learned

In this section, we interpret the results of our experiment and summarize our takeaways. Figure 4 depicts the clusters of submissions, where nodes represent submissions and edges represent direct equivalence proofs between submissions.

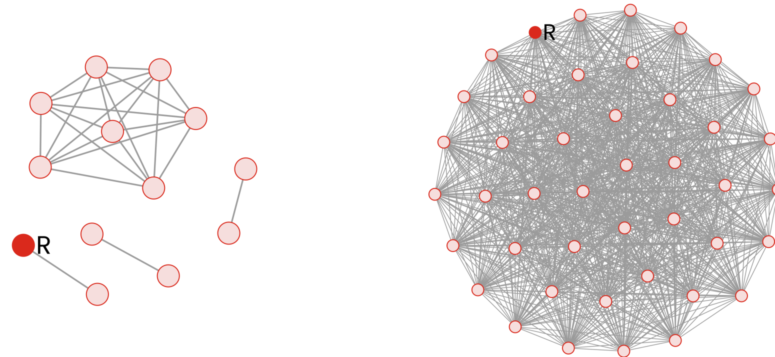
Solution Variations. We find that the verifier's success rate is lower than in the evaluation in an offline setting [39], with the majority of correct submissions timing out. We believe that this difference is due to the scarcity of reference solutions: in our experiment, we only provided one reference solution per exercise. Based on student feedback, this issue already became apparent during the semester. We addressed this problem by adding another reference solution for the `drop` exercise midway through the experiment, which improved the success rate.

Figure 4a shows the non-singleton clusters of submissions for the `drop` exercise. Our initial reference solution (node R) is in a cluster of size 7. Our additional reference solution (node R'), is in another cluster of size 17. The main difference between the two clusters is in counting to each n-th element, with solutions in the smaller cluster counting backwards from n down to 1 and solutions in the larger cluster counting forward from 1 to n. Another variation of the algorithm counts



(a) Clusters of submissions for the drop exercise. The initial reference solution (node R) is in a cluster of size 7. The second reference solution (node R') is in the largest cluster.

(b) Clusters of submissions for the gcd exercise. The reference solution from Figure 1a is in the smaller cluster (node R). Submissions from Figure 1b are in the larger cluster (nodes W, X, Y, Z).



(c) Clusters of submissions for the prime exercise. The submissions in the largest cluster compute non-optimized checks for division by each positive integer all the way to the input number.

(d) Clusters of submissions for the infix exercise, consisting of 8 non-recursive functions. All the submissions are in the same cluster because there is no recursion to differentiate the structure.

Fig. 4: Non-singleton clusters of submissions originally classified as correct or timed-out for all four exercises. Nodes represent submissions and edges represent direct equivalence proofs. The supplementary material contains .gexf files allowing further interactive inspection and graph manipulation via open-source tools such as https://gephi.org/gephi-lite/.

forward until the end of the list, computing each time the counter modulo n. This variation is problematic for potential overflows and for termination checks, although in practice we can assume that the size of the input list is smaller than Int.MaxValue.

We next consider the non-singleton clusters of submissions for the gcd exercise (Figure 4b). Programs from the smaller cluster (7 programs, 2 of which are shown in Figure 1a) use the subtraction-based Euclid's algorithm. Programs from the larger cluster (27 programs, 4 of which are shown in Figure 1b) use the modulo-based variation. Both clusters correspond to valid solutions for the gcd exercise. Furthermore, neither approach can be considered strictly better than the other. **Takeaway:** Even in introductory exercises, a single reference solution is not sufficient to capture the variety in student submissions. One should aim to provide a diverse set of reference solutions to reflect the diversity of student submissions.

Unique Solutions. Some submissions have a unique recursive structure and thus form a cluster of their own (column "Singleton" in Table 2). In the Ask-Elle studies [19], the authors devote particular attention to submissions that do not get matched against any reference solution and belong to a separate "correct (but no match)" category. Singleton clusters also appear in the Rainfall study [15], where they end up in a dedicated "other/unclear composition" category. We inspect the singleton clusters of submissions for the gcd exercise and identify the underlying causes for this classification:

- unique solution, identified by its unique recursion schema (Figure 1c)
- limitations of equivalence checking in Stainless (Figure 5a)
- limitations of formal equivalence checking (Figure 5b)

Despite passing all the tests, upon manual inspection, we were able to identify suboptimal implementations among those submissions. In introductory programming courses, such submissions could nevertheless be rewarded the maximum grade, but they are still worthy of custom feedback. On the other hand, in topics such as data science, where performance is of interest, such submissions should preferably be discouraged and only rewarded with partial points [53]. **Takeaway:** By failing to automatically classify certain supposedly correct submissions, the grader can reveal the submissions that require further manual inspection, and help detect suboptimal solutions.

The Role of Instructors. Our grader helped us identify not only unique solutions, but also whole clusters of suboptimal solutions. For example, consider the non-singleton clusters of submissions for the prime exercise (Figure 4c). Our reference solution (node R) checks if the input integer n is prime by dividing n by each positive integer up to the square root of n, or until a divisor is found. Interestingly, the most popular solution strategy among students turned out to be checking all the way to n instead.[3] Stainless could not prove this strategy

[3] This was also the case for the majority of submissions discarded due to compilation errors, implementing the same technique using for-comprehensions.

```
def gcd(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  def h(a : Int, b: Int, c: Int = b): Int =
    require(a >= 0 && b >= 0)
    if a < b then gcd(b,a)
    else if c == 0 then a
    else if b == 0 then a
    else if a == b then a
    else if (a%c==0)&&(b%c==0) then c
    else h(a, b, c−1)
  h(a,b)
```

(a) Limitations of Stainless. This program uses the same algorithm as the program in Figure 1c. Yet, the two programs did not end up in the same cluster, due to Stainless attempting at proving the equivalence by decomposing programs into equivalent functions. However, the two inner functions are unfortunately not equivalent, making it impossible for Stainless to conclude the proof.

```
def gcd(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if a == b then a
  else if a < b then gcd(b, a)
  else if a == 0 then b
  else if b == 0 then a
  else
    val r = a % b
    val q = (a − r) / (a / b)
    gcd(q, r)
```

(b) Limitations of formal equivalence checking. Due to an unusual choice of one student to recompute `b` as `(a − a%b) / (a/b)` on line 9, Stainless was unable to prove the equivalence of this submission to any other submission. Other equivalence checking tools, namely REVE [13] and RVT [21], also resulted in a timeout.

Fig. 5: Singleton clusters from the `gcd` exercise, pointing to submissions that require further manual inspection.

equivalent to our reference solution, forming a separate cluster of size 7. **Takeaway:** Every submission that results in a timeout (passes the tests but is not provably correct) should be checked by instructors. With the help of clustering, the number of manual checks reduces to one representative submission check per cluster. Upon manual inspection, correct submissions can be promoted to reference solutions. This way, the set of tests and reference solutions grows over time, to the point where only new singleton submissions are checked manually.

Library Functions. To tame the disparity between the Scala `List` class and the Stainless `List` class[4], we provide a stripped version of Stainless `List` with the handout, asking the students to use this version instead of Scala's. This way, we were able to exploit the benefits of formal verification without discouraging students from using library functions. For example, in the `drop` exercise, we observe submissions using library functions such as `foldLeft`, `length`, and `size`, each forming a separate cluster. One cluster of size two contains tail-recursive programs that use list concatenations. **Takeaway:** Some grading assistants introduce restrictions on using library functions [9]. Yet, support for library functions is crucial,

[4] The implementation of the `List` class in the Scala library internally mutates the tail for efficiency. To facilitate verification, the Stainless library provides two simpler implementations (an invariant and a covariant list) .

as it allows exercising functional programming abstractions. Library functions also lead to more diversity in student submissions.

Rigorous Autograding. Our students receive feedback for each (re)submission, allowing them to make progress from safety errors to logical errors, and finally to error-free programs. This descriptive feedback was well received by the students. Several students reported on the course forum that the grader found bugs in their code that they did not detect locally by running the test suite. For example, in the `gcd` exercise, 7 out of 8 incorrect submissions are due to safety errors in auxiliary functions, that could not be detected by the test suite. The authors of the LAV verifier make similar observations [61]. In their evaluation on C programming assignments, they show that, out of 266 submissions, LAV found 35 incorrect submissions that successfully passed the test suite, mainly due to subtle buffer overflows.

Whereas it is clear that programming assignments are not safety-critical on their own, teaching programming means teaching potential future authors of safety-critical software. Our study advocates for rigorous autograding, in a way that is completely transparent for students (push-button verification [44,58,40]). Namely, students are encouraged to think rigorously about program correctness, without the need for any in-depth knowledge about formal methods. This setting is in line with the vision of formal methods thinking in computer science education put forward by B. Dongol et al. [12] **Takeaway:** Formal verification enriches the feedback given to students. It provides feedback for errors that prevent correctness even if they cannot be formulated easily as counterexamples to program safety.

5 Limitations and Threats to Validity

In this section, we consider some limitations of our approach and our study in the context of practical automated grading.

Compilation Errors. We attribute large numbers of compilation errors to students only having local access to the Scala compiler, and not Stainless.[5] To reduce the number of compilation errors, we should allow students to run Stainless on their local machines. In each problem statement, we should clearly specify which language constructs are supported and which ones are forbidden. Ideally, we should provide local access to Stainless without safety and termination checks. This point is supported by previous research on the role of feedback, which suggests that introducing some form of delay is better for learning purposes, as immediate feedback is prone to undesirable trial-and-error solving strategies [7]. We notice similar concerns in a related report on the Learn-OCaml web platform [25], posing the question whether limiting the number of resubmissions would reduce the number of trivial and syntactic errors.

Syntax and Style. While it would be possible to further incorporate techniques for syntax and style checking [25,28], in this report, we focus on semantic pro-

[5] Stainless supports a subset of Scala (Section 2).

gram structure defined by recursion schemas. For example, the infix exercise is non-recursive, and therefore all the submissions are in the same cluster, despite students using a mixture of if-then-else, pattern matching, built-in boolean and bitwise operators, and custom (non-recursive) boolean functions. Furthermore, program clustering by recursion schemas does not differentiate purely functional programs from programs with loops and mutations. For example, submission gcdY with a while loop and variable mutation[6] is proven equivalent to purely functional submission gcdW, and thus placed in the same cluster (Figure 1b).

General Approach. Our analysis only uses data from 709 submissions from one course at our university, and does not necessarily represent the general trend in programming education. We found this setting (200 students, 4 exercises) to be a reasonable size for a pilot study, which enabled us to identify interesting aspects of in-class deployment.

While our study uses the Scala language and the Stainless verifier, our approach applies to other languages and verifiers. In particular, equivalence proofs by functional induction are applicable to recursive programs in general [37] (notable examples include functional induction in Coq [30], recursion induction [43] or computation induction [42] in Isabelle, default induction heuristic in ACL2 [32]). Furthermore, tools such as REVE [13], RVT [21] or SymDiff [33] have shown that equivalence checking is also feasible for imperative programs.

The nature of equivalence checking restricts our experiment to exercises with well-defined input-output behaviour, and does not cover open-ended problems.

6 Related Work

In this section, we describe related work on functional programming education, state-of-the-art clustering-based grading assistants, and applications of formal verification tools in programming courses.

Functional Programming Education. The Rainfall problem [55], originally studied in the field of imperative programming education, has recently become an insightful benchmark in functional programming education. In [15], the author takes over 200 solutions to the Rainfall problem across five functional-first courses and manually splits the solutions based on structure. The Rainfall studies later inspired evaluation of techniques for scaling program classification using machine learning, assuming that an instructor has indicated categories of interest [10]. In contrast, our approach automatically discovers clusters of submissions using equivalence proofs.

Learn-OCaml is an online grading platform for the OCaml MOOC [6]. Researchers have proposed extensions to the platform allowing assessment of style and test quality [25], as well as understanding how students interact with the grader [18]. Learn-OCaml's ability to keep track of metrics such as grades, the number of syntax errors and the time spent on each question enables clustering

[6] Stainless supports a limited form of side effects, such as mutation to mostly non-aliased state, and internally handles loops by transformations to recursive functions.

of students into four fixed interaction strategies. In contrast, we consider clustering of submissions, and dynamic discovery of clusters. It would be interesting to combine techniques from Lean-Ocaml with our approach and relate interaction strategies to underlying program structures.

Ask-Elle [19] is an online tutor for introductory Haskell exercises, providing feedback and incremental hints using property-based testing and strategy-based tracing. Like in our experiment, the case studies on Ask-Elle observe different patterns in student submissions, and show how Ask-Elle benefits from having multiple reference solutions that provide strategy-specific guidance. Custom feedback in Ask-Elle and Learn-OCaml comes at a cost of significant manual effort to set up a grader. In Ask-Elle, the instructor provides annotations for reference solutions and manually specifies QuickCheck [8] properties. Similarly, for each new Learn-OCaml exercise, the instructor has to specify custom syntax checks, predict unusual solutions, and write mutants to evaluate student-written tests. In contrast, in our deployment, we only had to provide a reference solution for each new exercise, along with optional MUnit tests.

Program Clustering in Grading Assistants. The idea of using equivalence checking to detect algorithmic similarity was previously explored in the ZEUS grading assistant [9]. Like our grader, ZEUS also relies on SMT solving, but while we use functional induction, ZEUS uses inference rules that simulate relationships between expressions. ZEUS is thus more restricted with respect to recursion and introduces limitations for programs with library functions. While both [9] and [39] focus on empirical evaluation of equivalence checking, the focus of our analysis is on *understanding* the resulting clusters and corresponding program structures.

OverCode [20] is a grading assistant for large scale courses, providing an interactive user interface for visualizing clusters of solutions. OverCode supports manual manipulation of program clusters, e.g., merging clusters by adding rewrite rules. However, unlike our approach, OverCode performs neither automated testing nor verification to check for program correctness. Neither OverCode nor ZEUS provide counterexamples for incorrect programs. Complementing our approach with OverCode's user interface and merge rules could be beneficial to ease manipulation of program clusters and counterexamples.

Verification Tools as Grading Assistants. LEGenT [1] is a tool for personalized feedback generation, using Clara [24] for program clustering and REVE [13] to identify provably correct submissions. LAV [61] is a verification tool evaluated on imperative programming assignments. Unlike our approach, which supports recursion, both LEGenT and LAV are targeting non-recursive programs.

Dracula [46,59,47] combines the ACL2 theorem prover [32] with the DrScheme graphical interface [14], in introductory programming and software engineering courses. The authors have used Dracula in undergraduate courses on functional programming and software engineering, like is the case in our case study. The main difference is that, in their setting, the goal is to tech students to formulate theorems (programming and proving). In contrast, in our approach, we do not go as far as asking students to systematically prove program properties, or

even state them. Instead, in our study, we provide a setting that encourages students to think rigorously about program correctness, without any in-depth knowledge about formal methods. Furthermore, unlike our approach, Dracula does not perform program clustering. Both ACL2 and Stainless have support for automated functional induction, which suggests that it would be possible to perform a similar study in ACL2, even if ACL2 is not higher-order.

Recently, researchers are increasingly sharing their experience on using proof assistants for teaching [3], both for mathematics and computer science programs. Proof assistants have been increasingly finding their way in specialized graduate courses [48,41,31], in undergraduate courses [29,17,50,16,36], and even high schools [23,4]. The question remains whether theorem provers can offer an adequate sufficiently high-level interface for students to write proofs without having to learn the proof assistant itself.

7 Conclusions

We have reported our experience in using a formal verifier for evaluation of assignments in an undergraduate programming course. We found that formal verification enriches the feedback given to students. Moreover, verification based on functional induction allowed us to differentiate between solutions, even when solutions exhibit the same input-output behaviour. It allowed us to propose additional reference solutions, and to focus our attention on unusual solutions. We are therefore confident that this approach represents a useful addition to automating assignment evaluation.

We have shared our main takeaways and our deployment process in detail, with the hope that our study will provide inspiration for others trying to incorporate program verifiers and program clustering in assignment assessment in their own courses. In the future, we will progressively use the approach on more exercises of our course. To improve the quality of feedback reported to students, we will continuously grow the set of reference solutions and provide more refined verification outcome summaries.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Agarwal, N., Karkare, A.: LEGenT: Localizing Errors and Generating Testcases for CS1. In: Proceedings of the Ninth ACM Conference on Learning @ Scale. p. 102–112. L@S'22, Association for Computing Machinery (2022). https://doi.org/10.1145/3491140.3528282
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442 (2022). https://doi.org/10.1007/978-3-030-99524-9_24

3. Bartzia, E., Meyer, A., Narboux, J.: Proof assistants for undergraduate mathematics and computer science education: elements of a priori analysis. In: Trigueros, M. (ed.) INDRUM 2022: Fourth conference of the International Network for Didactic Research in University Mathematics. Reinhard Hochmuth, HAL, Hanovre, Germany (Oct 2022), https://hal.science/hal-03648357
4. Bertot, Y., Guilhot, F., Pottier, L.: Visualizing Geometrical Statements with GeoView. Electr. Notes Theor. Comput. Sci. **103**, 49–65 (11 2004). https://doi.org/10.1016/j.entcs.2004.09.013
5. Blanc, R.W.: Verification by Reduction to Functional Programs p. 191 (2017). https://doi.org/https://doi.org/10.5075/epfl-thesis-7636
6. Canou, B., Di Cosmo, R., Henry, G.: Scaling up functional programming education: under the hood of the OCaml MOOC. Proc. ACM Program. Lang. **1**(ICFP) (aug 2017). https://doi.org/10.1145/3110248
7. Chevalier, M., Giang, C., El-Hamamsy, L., Bonnet, E., Papaspyros, V., Pellet, J.P., Audrin, C., Romero, M., Baumberger, B., Mondada, F.: The role of feedback and guidance as intervention methods to foster computational thinking in educational robotics learning activities for primary school. Computers & Education **180**, 104431 (2022). https://doi.org/10.1016/j.compedu.2022.104431
8. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. p. 268–279. ICFP '00 (2000). https://doi.org/10.1145/351240.351266
9. Clune, J., Ramamurthy, V., Martins, R., Acar, U.A.: Program Equivalence for Assisted Grading of Functional Programs. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428239
10. Crichton, W., Sampaio, G.G., Hanrahan, P.: Automating Program Structure Classification. In: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education. p. 1177–1183. SIGCSE '21 (2021). https://doi.org/10.1145/3408877.3432358
11. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 337–340. TACAS'08/ETAPS'08 (2008), https://doi.org/10.1007/978-3-540-78800-3_24
12. Dongol, B., Dubois, C., Hallerstede, S., Hehner, E., Morgan, C., Müller, P., Ribeiro, L., Silva, A., Smith, G., de Vink, E.: On Formal Methods Thinking in Computer Science Education. Form. Asp. Comput. (2024). https://doi.org/10.1145/3670419, https://doi.org/10.1145/3670419, just Accepted
13. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating Regression Verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 349–360. ASE '14 (2014). https://doi.org/10.1145/2642937.2642987
14. Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: a programming environment for Scheme. J. Funct. Program. **12**(2), 159–182 (2002). https://doi.org/10.1017/S0956796801004208
15. Fisler, K.: The recurring rainfall problem. In: Proceedings of the Tenth Annual Conference on International Computing Education Research. p. 35–42. ICER'14 (2014). https://doi.org/10.1145/2632320.2632346
16. From, A., Jacobsen, F., Villadsen, J.: SeCaV: A sequent calculus verifier in Isabelle/HOL. In: Proceedings of 16th Logical and Semantic Frameworks with Applications. Electronic Proceedings in Theoretical Computer Science, EPTCS, vol. 357, pp. 38–55 (2022). https://doi.org/10.4204/EPTCS.357.4

17. Gambhir, S., Guilloud, S., Milovančević, D., Rümmer, P., Kunčak, V.: Lisa tool integration and education plans (2023)
18. Geng, C., Xu, W., Xu, Y., Pientka, B., Si, X.: Identifying Different Student Clusters in Functional Programming Assignments: From Quick Learners to Struggling Students. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education. p. 750–756. SIGCSE 2023 (2023). https://doi.org/10.1145/3545945.3569882
19. Gerdes, A., Heeren, B., Jeuring, J., van Binsbergen, L.T.: Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. International Journal of Artificial Intelligence in Education **27** (2016). https://doi.org/10.1007/s40593-015-0080-x
20. Glassman, E.L., Scott, J., Singh, R., Guo, P.J., Miller, R.C.: Overcode: Visualizing variation in student solutions to programming problems at scale. ACM Trans. Comput.-Hum. Interact. **22**(2) (2015). https://doi.org/10.1145/2699751, https://doi.org/10.1145/2699751
21. Godlin, B., Strichman, O.: Regression verification: Proving the equivalence of similar programs. Software Testing Verification and Reliability **23**, 241–258 (2013). https://doi.org/10.1002/stvr.1472
22. Griswold, W.G.: Experience Report: Meet the Professor - A Large-Course Intervention for Increasing Rapport. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. p. 415–421. SIGCSE 2024 (2024). https://doi.org/10.1145/3626252.3630844
23. Guilhot, F.: Formalisation en Coq et visualisation d'un cours de géométrie pour le lycée. Technique et Science Informatiques **24**, 1113–1138 (2005). https://doi.org/10.3166/tsi.24.1113-1138
24. Gulwani, S., Radiček, I., Zuleger, F.: Automated clustering and program repair for introductory programming assignments. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 465–480. PLDI 2018 (2018). https://doi.org/10.1145/3192366.3192387, https://doi.org/10.1145/3192366.3192387
25. Hameer, A., Pientka, B.: Teaching the art of functional programming using automated grading (experience report). Proc. ACM Program. Lang. **3**(ICFP) (2019). https://doi.org/10.1145/3341719
26. Hamza, J., Felix, S., Kunčak, V., Nussbaumer, I., Schramka, F.: From Verified Scala to STIX File System Embedded Code Using Stainless. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods. pp. 393–410. Springer International Publishing, Cham (2022). https://doi.org/https://doi.org/10.1007/978-3-031-06773-0_21
27. Hamza, J., Voirol, N., Kunčak, V.: System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360592
28. Hart, R., Hays, B., McMillin, C., Rezig, E.K., Rodriguez-Rivera, G., Turkstra, J.A.: Eastwood-Tidy: C Linting for Automated Code Style Assessment in Programming Courses. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. p. 799–805. SIGCSE 2023 (2023). https://doi.org/10.1145/3545945.3569817
29. Henz, M., Hobor, A.: Teaching Experience: Logic and Formal Methods with Coq. In: Jouannaud, J.P., Shao, Z. (eds.) Certified Programs and Proofs. pp. 199–215. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_16

30. INRIA: Functional Induction in Coq. https://coq.inria.fr/refman/using/libraries/funind.html (2021)
31. Jacobsen, F., Villadsen, J.: On exams with the isabelle proof assistant. Electronic Proceedings in Theoretical Computer Science **375**, 63–76 (03 2023). https://doi.org/10.4204/EPTCS.375.6
32. Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, USA (2000), https://doi.org/10.1007/978-1-4615-4449-4
33. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: Proceedings of the 24th International Conference on Computer Aided Verification. p. 712–717. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_54
34. LARA: Stainless. https://github.com/epfl-lara/stainless (2023)
35. Lee, J., Song, D., So, S., Oh, H.: Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments. Proc. ACM Program. Lang. **2**(OOPSLA) (2018). https://doi.org/10.1145/3276528
36. Maxim, H., Kaliszyk, C., van Raamsdonk, F., Wiedijk, F.: Teaching logic using a state-of-art proof assistant. Acta Didactica Napocensia **3** (2010)
37. McCarthy, J.: A basis for a mathematical theory of computation. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, Studies in Logic and the Foundations of Mathematics, vol. 35, pp. 33–70. Elsevier (1963). https://doi.org/10.1016/S0049-237X(08)72018-4
38. Messer, M., Brown, N.C.C., Kölling, M., Shi, M.: Automated Grading and Feedback Tools for Programming Education: A Systematic Review. ACM Trans. Comput. Educ. **24**(1) (2024). https://doi.org/10.1145/3636515
39. Milovančević, D., Kunčak, V.: Proving and Disproving Equivalence of Functional Programming Assignments. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591258
40. Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., Wang, X.: Hyperkernel: Push-Button Verification of an OS Kernel. In: Proceedings of the 26th Symposium on Operating Systems Principles. p. 252–269. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3132747.3132748
41. Nipkow, T.: Teaching semantics with a proof assistant: No more lsd trip proofs. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 24–38. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
42. Nipkow, T.: Programming and Proving in Isabelle/HOL. https://isabelle.in.tum.de/dist/Isabelle2022/doc/prog-prove.pdf (2022)
43. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002), https://doi.org/10.1007/3-540-45949-9
44. Oberhauser, J., Chehab, R.L.d.L., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., Vafeiadis, V.: Vsync: push-button verification and optimization for synchronization primitives on weak memory models. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 530–545. ASPLOS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3445814.3446748

45. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, Fourth Edition (A comprehensive step-by-step guide). Artima, Sunnyvale, CA, USA (2019), https://www.artima.com/shop/programming_in_scala_4ed
46. Page, R.: Engineering software correctness. In: Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education. p. 39–46. FDPE'05 (2005). https://doi.org/10.1145/1085114.1085123
47. Page, R., Eastlund, C., Felleisen, M.: Functional programming and theorem proving for undergraduates: a progress report. In: Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education. p. 21–30. FDPE'08 (2008). https://doi.org/10.1145/1411260.1411264
48. Pierce, B.C.: Lambda, the ultimate ta: using a proof assistant to teach programming language foundations. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. p. 121–122. ICFP '09 (2009). https://doi.org/10.1145/1596550.1596552, https://doi.org/10.1145/1596550.1596552
49. Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: Sk_p: A neural program corrector for moocs. In: Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity. p. 39–40. SPLASH Companion 2016 (2016). https://doi.org/10.1145/2984043.2989222
50. Rousselin, P.: Mathematics with Coq for first-year undergraduate students (2023)
51. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Logic for Programming, Artificial Intelligence, and Reasoning. pp. 274–289 (2008). https://doi.org/https://doi.org/10.1007/978-3-540-89439-1_20
52. Schmid, G.S., Kuncak, V.: Generalized arrays for stainless frames. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13182, pp. 332–354. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_17
53. Singh, A., Fariha, A., Brooks, C., Soares, G., Henley, A.Z., Tiwari, A., M, C., Choi, H., Gulwani, S.: Investigating student mistakes in introductory data science programming. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. p. 1258–1264. SIGCSE 2024 (2024). https://doi.org/10.1145/3626252.3630884
54. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. SIGPLAN Not. **48**(6), 15–26 (2013). https://doi.org/10.1145/2499370.2462195
55. Soloway, E.: Learning to program = learning to construct mechanisms and explanations. Commun. ACM **29**(9), 850–858 (1986). https://doi.org/10.1145/6592.6594
56. Song, D., Lee, M., Oh, H.: Automatic and scalable detection of logical errors in functional programming assignments. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360614
57. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) Static Analysis. pp. 298–315 (2011). https://doi.org/10.1007/978-3-642-23702-7_23
58. Tao, R., Shi, Y., Yao, J., Li, X., Javadi-Abhari, A., Cross, A.W., Chong, F.T., Gu, R.: Giallar: push-button verification for the qiskit quantum compiler. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 641–656. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523431

59. Vaillancourt, D., Page, R., Felleisen, M.: ACL2 in DrScheme. In: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications. p. 107–116. ACL2'06 (2006). https://doi.org/10.1145/1217975.1217999
60. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Proceedings of the 6th ACM SIGPLAN Symposium on Scala. p. 18–29. SCALA 2015 (2015). https://doi.org/10.1145/2774975.2774978
61. Vujošević-Janičić, M., Nikolić, M., Tošić, D., Kuncak, V.: Software verification and graph similarity for automated evaluation of students' assignments. Inf. Softw. Technol. **55**(6), 1004–1016 (2013). https://doi.org/10.1016/j.infsof.2012.12.005
62. Wang, K., Singh, R., Su, Z.: Search, align, and repair: Data-driven feedback generation for introductory programming exercises. SIGPLAN Not. **53**(4), 481–495 (2018). https://doi.org/10.1145/3296979.3192384
63. Wrenn, J., Krishnamurthi, S., Fisler, K.: Who tests the testers? In: Proceedings of the 2018 ACM Conference on International Computing Education Research. p. 51–59. ICER'18, Association for Computing Machinery (2018). https://doi.org/10.1145/3230977.3230999