Formal Autograding in a Classroom (Experience Report)

ANONYMOUS AUTHOR(S)

We report our experience in enhancing automated grading in a functional programming course using formal verification. In our approach, we deploy a verifier for Scala programs to check equivalences between student submissions and reference solutions. Consequently, students receive more thorough evaluation of assignments that explores behaviours beyond those envisioned by tests or test generators. We collect student submissions and make them publicly available. We analyse the collected data set and find that we can use the conservative nature of our program equivalence checking as an advantage: we were able to use such equivalence to differentiate student solutions according to their high-level program structure, in particular their recursion pattern, even when their input-output behaviour is identical.

1 INTRODUCTION

With the growing numbers of students in programming courses, automated grading of student assignments has become ubiquitous. The goal of practical and rigorous automated grading has motivated the development of tools to aid teaching programming courses [Agarwal and Karkare 2022; Gulwani et al. 2018; Lee et al. 2018; Pu et al. 2016; Singh et al. 2013; Song et al. 2019, 2021; Wang et al. 2018]. These tools commonly employ automated testing to evaluate the correctness of student submissions [Messer et al. 2024].

We observed this general trend in the functional programming course at our university. Whereas seven years ago the course counted around 200 students, this number has more than doubled since then. The course contains weekly programming exercises of various difficulty, as well as a written midterm exam and a computer-based final exam. To maintain a reasonable work load for the teaching staff, the course heavily relies on testing to automate the assessment of student programs throughout the semester. We similarly grade the final exam using a test suite, with manual inspection reserved for ambiguous cases.

While highly automated, such testing-based grading comes at a cost of accuracy and feedback quality. Researchers have shown that flaws in test suites, and the resulting miss-classifications of student programs, can have negative impact on the students [Wrenn et al. 2018]. Furthermore, among the solutions that do satisfy the input-output requirements, testing-based automated graders fail to provide solution-specific feedback [Clune et al. 2020; Gerdes et al. 2016].

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. Yet these are solutions that deserve special attention and custom feedback. As an example, consider the programs from Figure 1, computing gcd for two natural numbers. The program from Figure 2 exhibits exactly the same input-output behaviour as the programs from Figure 1, despite being unique and less efficient. This solution would unfortunately typically go unnoticed with a testing-based grader, despite its unique underlying approach. Furthermore, while all the programs from Figure 1 have identical input-output behaviour, we can identify two clusters with different underlying approaches (subtraction-based vs. modulo-based Euclid's algorithm). We would like our autograder to identify the two solution approaches, and cluster the programs not just by correctness, but also by the underlying approach. Specifically, we are interested in the semantic and not syntactic level of granularity such that all the programs from Figure 1b are in the same cluster and the program from Figure 2 is in a different cluster.

In this paper, we explore the use of formal verification tools for automated grading, aiming to improve the accuracy over existing testing-based automation. 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. Our main inspiration is the Rainfall

```
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) Our reference solution (top) and one student submission from the same cluster (bottom).

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


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 submissions from another (larger) cluster, neither of them proven equivalent to our reference solution. The four programs are proven equivalent among themselves, either directly or via another submission.

Fig. 1. Two clusters of solutions for the gcd exercise, illustrating two different recursion schemas.

study from functional programming education [Fisler 2014], and its impact to understanding the importance of variety in students' programs. However, the analysis from [Fisler 2014] is a manual, time intensive process, and consists of first identifying program clusters and then assigning labels to each individual program. Whereas human insight is indispensable, this paper explores the potential for automatically finding such labels, e.g. representatives from Figure 1 for different clusters.

Focusing on functional programming assignments, 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 verifier for Scala, which was previously evaluated on several thousands programming assignments in an offline setting [Milovančević and Kunčak 2023]. In that work, the authors use Stainless to verify program correctness against reference solutions, using program clustering and formal equivalence checks. In contrast, we report on our experience using Stainless as a verification-based grader in the *live setting* of a functional programming course. We perform analysis of solutions beyond their correctness, clustering student submissions independently of the reference solutions. We find that the clusters of equivalent submissions provide useful insights into different program structures, beyond those envisioned by the provided reference solution.

The main contributions of this paper are:

- We report on our experiment with deploying the Stainless verifier as an automated grader for introductory functional programming exercises in our course. We explain our deployment process in detail and share our insights.
- We show that formal verification can reveal variations in the underlying approach 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 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)
```

Fig. 2. One of 12 singleton submissions from the gcd exercise. Observe the difference in its recursive calls compared to submissions in the two main clusters in Figure 1.

```
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. 3. An incorrect student submission from the drop exercise. Stainless detects a concrete counterexample, which we report as feedback to the student.

2 BACKGROUND: STAINLESS FORMAL VERIFIER

One differentiating characteristic of our study is the use of a formal verifier. Our course teaches Scala [Odersky et al. 2019], so we adopt the Stainless verifier [LARA 2023]. Stainless can prove program termination and the absence of runtime failures such as division by zero or pattern matching exhaustivity. Developers can optionally provide program specification using contracts, such as pre- and post-conditions on functions, invariants on classes and loops, and assertions. Contracts are expressed within the syntax of the Scala language, such as **assert**, **require**, and **ensuring** statements. Stainless supports only a subset of Scala, with best support for a purely functional subset with ML-style polymorphism. This subset aligns well with the Scala subset that we present in our course and expect students to use in course assignments.

Stainless works by first parsing and type checking the input program using the Scala compiler to obtain an abstract syntax tree (AST). It applies transformations on the AST to obtain an equivalent purely functional program, from which it generates verification conditions using a type checking algorithm [Hamza et al. 2019]. It iteratively unfolds recursive functions [Suter et al. 2011; Voirol et al. 2015] and uses SMT solvers (Z3 [De Moura and Bjørner 2008], CVC5 [Barbosa et al. 2022], Princess [Rümmer 2008]) to either prove or disprove those verification conditions. Our deployment makes use of a high-level functionality of Stainless to perform equivalence checking of programs [Milovančević and Kunčak 2023]. In this mode, rather than writing pre- and post-conditions, the user provides specification in the form of a reference program. Stainless then attempts to prove program correctness via automated equivalence proofs against the reference program. Our deployment also benefits from the ability of Stainless to generate counterexamples for incorrect programs [Voirol et al. 2015], providing feedback that is valuable to both students and instructors (Figure 3).

Table 1. Description of our exercises. #P indicates the number of submissions per exercise. #F and LOC show the average number of functions per program and the average number of lines of code, respectively.

No	Name	Description	#P	#F	LOC
1	gcd	Computing the greatest common divisor of two integers	80	1.3	11
2	drop	Dropping every n-th element from a list	373	1.8	13
3	prime	Checking if an integer is prime or not	220	3.7	19
4	infix	Implementing infix operators for booleans	46	7.8	17

3 THE EXPERIMENT

In this section, we report on our experiment adapting and 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 the basics of functional programming in Scala, 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 graded 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, comprising smaller problems designed to help grasp the course material, such as evaluation of algebraic expressions, manual recursion elimination, and memoisation.

3.2 Experimental Setup

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 their solutions. We collected the data following a protocol approved by our university's human research ethics committee. The code was automatically graded by running 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 described in Section 4.2. For each submission, students received automated feedback that they could inspect. They were permitted to re-submit solutions any number of times.

At the end of the course, we additionally examined the submitted programs, using Stainless as a grading assistant on the instructor side. To this end, we gathered all the submissions proven correct, together with the reference solutions, and all the submissions where 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 student submissions.

3.3 Setting Up Stainless as a Grader

We next describe how we adapted Stainless 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 on-premise Kubernetes cluster service evaluates the assignments by running specified tests and uploading the grade and the feedback for students 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 the feedback to students. We create a dedicated Docker image for each exercise, packaged with Stainless and Z3. An orchestrating script is responsible for collecting submissions, feeding them into Stainless along with the reference solutions, and producing feedback from the output of Stainless.

Exercise Setup. We prepare our exercises following the template for the existing course assignments. To illustrate the usability of our approach, we detail the steps needed to prepare an exercise:

(1) Write the problem statement as a markdown file and upload it to the course website.
(2) Set up a dedicated SBT Scala project with the template Scala source files.
(3) Write the reference solution(s), without any additional specification or annotations.
(4) Write MUnit tests to allow students to test their solutions locally without needing Stainless.
(5) Write a configuration file describing the name of the function(s) and the reference solution(s) for the equivalence checking.
(6) Pack everything into a Docker image and link it to a dedicated Moodle assignment.

Emulating a Subset of Scala's Standard Library. Some of our exercises use Scala's `List` class, whose implementation internally mutates the tail for efficiency. To make verification feasible, Stainless library provides two simpler implementations of `List` as alternatives: a type-parameter invariant list, and a type covariant list. When covariance is not necessary, the invariant version is preferred since the generated verification conditions can be solved more efficiently due to more direct mapping to algebraic data types of SMT solvers. Furthermore, Stainless and Scala signatures do not agree for certain `List` methods, such as the indexing operation `apply` taking a `BigInt` for Stainless but an `Int` for Scala. To tame such disparity, we provide a stripped version of Stainless `List` with the handout, asking the students to use this version instead of Scala's `List`.

Feedback Generation. For each submission, students receive the feedback consisting of a boolean success grade, a text file with comments for each function, and a log file for further inspection, including a detailed log output of Stainless. The boolean success grade is either full credits when the test, safety, and equivalence checks successfully passed, or zero credits if either of those failed. We report the outcome of additional termination checks that do not influence the grade. The success grade does not affect student's course performance and is merely a summary feedback for students. The textual feedback file contains the more detailed information. It can either contain a congratulation message, in case the submission is provably correct, or otherwise an explanation of the encountered error, such as in our example shown in Figure 3. More precisely, we report custom feedback for the following errors:

– **User errors** – such as an incorrectly named file or an incorrect function signature
– **Safety check errors** – such as division by zero or integer overflow
– **Counterexample errors** – a counterexample input was found
– **Termination errors** – a function could not be proven to be terminating
– **Equivalence errors** – a function could not be proven equivalent to any reference solution

The feedback helps students iteratively improve their code until they solve each problem.

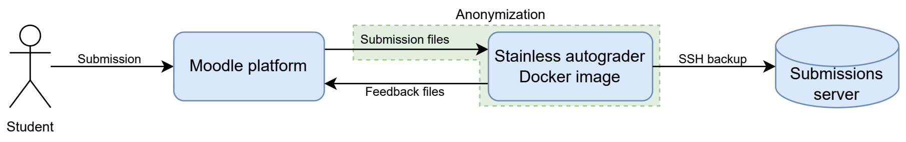


Fig. 4. Data collection pipeline.

Data Collection. Figure 4 illustrates our data collection pipeline. Upon each assignment submission, the submission data is automatically anonymized and copied out. We collect the submitted Scala files, generated feedback files, and debug logs. Each collected sample is initially identified by a version 4 UUID, which we also include on the Moodle side to create a one-way link from the submissions to the corresponding samples. This link allows us to inspect issues and to remove collected samples in case a student opts-out of participation in the study. Students were instructed to not leave any personal information in submission files. Other than the contents of submitted files, data collection does not persist any information about the student or the submission.

4 RESULTS AND LESSONS LEARNED

In this section, we introduce our data set (Section 4.1), and present the results and lessons learned from using formal equivalence checks in our course (Section 4.2). We then present the main clustering results (Section 4.3). We found the resulting clusters insightful for understanding the variations in solution strategies. As a result, we were able to identify representative solutions that can serve as good initial reference solutions for future iterations of the course.

4.1 New Public Data Set for Functional Programming Assignments

Table 1 shows the total number of submissions per exercise, along with the average number of functions per submission and average number of lines of code per submission. Exercises gcd, drop and prime are recursive. The infix exercise consists of 8 non-recursive functions.

Our data set comprising 709 Scala programs is available as supplementary material with 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. In [Messer et al. 2024], the authors analyse 121 research papers in the field and remark that, indeed, only 10 of them have publicly available data sets.[1]

4.2 Basic Analysis of Results and Lessons Learned

We export the generated feedback and present the raw 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. 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.

Compilation Errors. Around one third of the submissions are not of interest due to compilation errors (Column S), resulting in around 500 syntactically valid programs. These errors are in part due to students only having local access to the Scala compiler, and not Stainless. **Lesson:** In the problem statement, we should specify which language constructs are supported and which ones are not. We

[1]We are grateful to our university's ethics committee and colleagues for their help throughout the process and our students for allowing us to publicly share their submissions. The overall process involved a significant administrative overhead, which possibly explains the scarcity of publicly available data sets in the literature on programming education.

Table 2. Results of our experiment, when providing a single reference solution per exercise. S indicates the submissions that could not be processed due to a compilation error, or due to students submitting wrong files. TS indicates submissions where Stainless could not prove safety checks. I and C show the number of submissions proven incorrect and correct, respectively. TO indicates programs where the equivalence check times out.

Name	S	TS	I	C	TO
gcd	21	2	8	3	42
drop	60	56	169	14	70
prime	146	9	40	1	22
infix	2	0	2	42	0

Table 3. Results of clustering by recursive program structure. We combine the C and TO submissions from Table 2 with the initial reference solutions to compute clusters of provably equivalent submissions. Column "Non-Singleton" shows the numbers of non-singleton clusters (illustrated in Figure 5). Column "Singleton" shows the numbers of the remaining singleton clusters (examples listed in Figure 2 and Figure 6).

Name	Non-Singleton	Singleton
gcd	2	12
drop	10	26
prime	4	11
infix	1	0

should allow students to run Stainless on their local machines, ideally without various safety and termination checks. This last point is supported by previous research on the role of feedback, which shows that introducing a minimal delay is better for learning, as immediate feedback is prone to undesirable trial-and-error solving strategies [Chevalier et al. 2022]. We notice similar concerns in a related experience report on the Learn-OCaml web platform [Hameer and Pientka 2019], further posing the question if limiting the number of resubmissions would reduce the number of trivial and syntactic mistakes.

Nature of Feedback. Our students get textual feedback for each (re)submission, allowing them to make progress from safety errors to logical errors (column I), and finally to counterexample-free programs (column TO), in some cases also proven correct (column C). 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 were not detected locally by the test suite. They could inspect the outcome of each submission in detail and iterate on their solutions. However, while focusing on the textual feedback, we underestimated the impact of numerical grades on undergraduate students [Kyrilov and Noelle 2015]. Our decision to keep the numerical grade at zero unless the program is proven correct had a discouraging effect for some students. **Lesson:** We should be more generous with partial points, in particular for timeout submissions where no counterexample was found.

Success Rate. Focusing on the submissions that our verifier proved correct (Column C), we find that the verifier's success rate is lower than in the evaluation in [Milovančević and Kunčak 2023], 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 for each exercise. **Lesson:** We should aim to provide a diverse set of reference solutions to reflect the diversity of student submissions. Thanks to students' discussions, 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.

In the next subsection, we present our clustering analysis, where we further explore the benefits of multiple reference solutions, as well as strategies for selecting good reference solutions.

4.3 Clustering Analysis

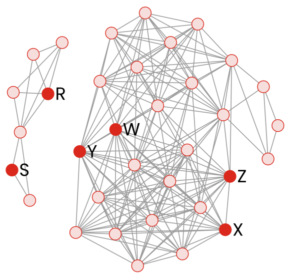
In this final chapter of our analysis, we use equivalence checking to cluster submissions based on their underlying recursive structure. The idea of using equivalence checking to detect algorithmic similarity was previously explored in Zeus [Clune et al. 2020]. While both [Clune et al. 2020] and [Milovančević and Kunčak 2023] focus on evaluating their equivalence checking tools, the focus of our analysis is on *understanding* the resulting clusters and corresponding program structures.

Our clustering analysis is subdivided per exercise. We consider a pool of programs consisting of the reference solutions, all the submissions where the equivalence proof timed out against the reference solution (column TO in Table 2), and all the submissions proven correct (column C in Table 2). For each exercise, the analysis consists in running the equivalence checking for each pair of programs, with a five-second solver timeout. We form clusters based on the results of these equivalence checks. Figure 5 depicts those clusters, where each node represents a submission and each edge represents a direct equivalence proof between submissions. Submissions that are not provably equivalent to any other submission form a singleton cluster of their own (omitted from Figure 5). We notice that our initial reference solutions (nodes R) did not end up in the largest cluster for either of the three non-recursive exercises. Together with the results from Section 4.2, the clusters in Figure 5 further suggest that, even in introductory exercises, there is more than a single approach to solve those problems. Table 3 shows the number of singleton and non-singleton clusters, per exercise. To further investigate the various approaches employed by our students, we next inspect the resulting program clusters for each exercise.

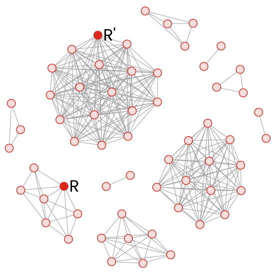
The gcd Exercise. Figure 5a shows the non-singleton clusters of gcd submissions. Some of those submissions appear in full in Figure 1 (nodes S, W, X, Y, Z); we further examine those programs in what follows. We can clearly identify two main underlying algorithms used by students. Programs from the smaller cluster (7 programs, 2 of which are shown in Figure 1a) use 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 of those clusters correspond to valid solutions for the gcd exercise. Furthermore, neither approach can be considered strictly better than the other.

Node W (which corresponds to gcdW in Figure 1b) has the highest degree, with direct edges to 19 other submissions in its cluster. This makes it a great candidate for an additional reference solution, even better than our initial reference solution whose degree is only four. Node X (which corresponds to gcdX in Figure 1b) does not have the require statement, which makes it a bad candidate for a reference solution. The reason is that, while this program does have the correct input-output behaviour for the defined input domain, it would wrongly discriminate other correct solutions with the require statement, for negative inputs. This submission in fact exposes a limitation of Stainless, which in the batch mode uses a transitivity-based algorithm to extend the set of reference solutions, without considering programs with preconditions. In the smaller cluster, the distance between nodes S and R (gcdS and gcdR Figure 1a) is two edges: there is one intermediate student submission. This distance reflects the slight difference in the recursive branches. Doubling the timeout for SMT queries results in a successful direct equivalence proof.

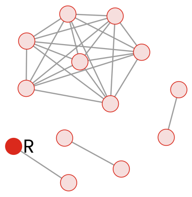
The drop Exercise. Figure 5b shows the non-singleton clusters of drop submissions. Most solutions delegate the element removal to an inner non-tail recursive function. Our initial reference solution (node R) is in a cluster of size 7. Mid-experiment, we provided an additional reference solution (node R'), which is in another cluster of size 17. The main difference is in counting to each n-th element, with solutions in the smaller cluster counting repetitively backwards from n down to 1 and solutions in the larger cluster counting repetitively forward from 1 to n. We notice further small variations in those looping intervals, e.g., inner function counting from (or stopping at) 0 vs 1 ($n - 1$ vs n). Another variation of the algorithm counts forward until the end of the list, computing each



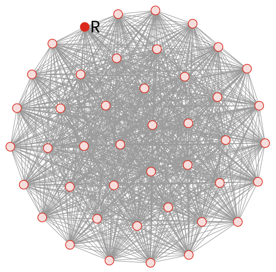
(a) Clusters of submissions for the gcd exercise. The initial reference solution from Figure 1a is in the smaller cluster (node R). The four student submissions from Figure 1b are in the larger cluster (nodes W, X, Y, Z).



(b) Clusters of submissions for the drop exercise. The initial reference solutions is in the cluster of size 7 (node R). The second reference solution is in the largest cluster (node R'). The difference is in counting to n forward or backwards.



(c) Clusters of submissions for the prime exercise. The largest cluster computes 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 eight non-recursive functions. All the submissions are in the same cluster because there is no recursion to define the structure.

Fig. 5. Clusters of all submissions originally classified as correct or timed-out for all four exercises. 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/.

time modulo n. This variation is problematic for potential overflows and for termination checks, although in practice we can count on the size of the input list to be smaller than MaxValue. Only one cluster of size two contains tail-recursive programs that use list concatenations. Finally, we observe a few unique solutions with other library functions such as foldLeft, length, or size, each forming a separate singleton cluster.

The prime Exercise. Figure 5c shows the non-singleton clusters of prime submissions. Interestingly, the most popular solution strategy turned out to be simply checking for division by each positive integer all the way to the input number. This was also the case for the majority of submissions discarded due to syntax errors, implementing the same technique using for-comprehensions that build on unsupported ranges. Smaller pair clusters are optimized variations, counting only up to the square root of the input number, using the provided library function isqrt.

The infix Exercise. Figure 5d shows the single cluster of infix submissions. The infix exercise is non-recursive, and therefore all the programs are in the same cluster. Manual inspection reveals a mixture of if-then-else, pattern matching, built-in boolean and bitwise operators, and custom boolean functions, implemented in the rest of the exercise. While it would be possible to further

```
def gcd(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  def helper(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 helper(a, b, c−1)
  helper(a,b)
```

(a) Singleton cluster illustrating limitations of function matching in Stainless.

```
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) Singleton cluster illustrating limitations of equivalence checking.

Fig. 6. Singleton clusters from the gcd exercise, exposing limitations of our equivalence-based clustering.

cluster submissions based on syntax and style, e.g., applying techniques for style checking such as in [Hameer and Pientka 2019], in this report, we focus on semantic program structure.

Singleton Clusters. Some submissions have a unique recursive structure and thus form a cluster of their own. Singleton clusters also appear in the Rainfall study [Fisler 2014], where they end up in a dedicated "other/unclear composition" category. In the Ask-Elle studies [Gerdes et al. 2016], although there is no explicit notion of clustering, the authors devote particular attention to submissions that do not get matched against any reference solution and belong to a dedicated "correct (but no match)" category. We inspect the source code of submissions in singleton clusters and identify the underlying causes that lead to this classification:

- unique solution strategy, identified by a unique recursion schema;
- limitations of Stainless, in particular for programs with inner functions;
- limitations of formal equivalence checking.

As an example of unique solution strategy, consider the gcd submission from Figure 2. This program iterates from the minimum of its arguments, one by one, until it finds a common divisor. Other submissions (shown in Figure 1) iterate faster: they all have different recursion schemas.

To illustrate a limitation of the equivalence checking in Stainless, consider the submission in Figure 6a. This program uses the same inefficient algorithm as the singleton cluster in Figure 2, including the same recursive calls. Yet, the two programs did not end up forming a joint cluster. The reason is that Stainless, in its current implementation, attempts at proving the equivalence by decomposing programs into equivalent helper functions. However, in this case, the two inner functions helper and checkGcd are unfortunately not equivalent, making it impossible for Stainless to conclude the proof.

To illustrate a limitation of formal equivalence checking, consider the submission in Figure 6b. This program has a general structure similar to the modulo-based programs in Figure 1b. However, due to a rather interesting optimization in this submission (line 9), Stainless was unable to prove the equivalence. Furthermore, we consulted other specialised equivalence checking tools that support recursion, namely REVE [Felsing et al. 2014] and RVT [Godlin and Strichman 2013]. Both resulted in a timeout, suggesting that this submission exposes a general limitation of formal equivalence checking.

5 RELATED WORK

We first report on how our work relates to prior studies on functional programming education. We then relate our work to state-of-the-art clustering-based grading assistants. Finally, we touch on the subject of teaching programming with the help of formal techniques, using program verifiers and proof assistants.

Related Literature on Functional Programming Education. The Rainfall problem [Soloway 1986], originally studied in the field of imperative programming education, has recently become an insightful benchmark in the context of teaching functional programming. In [Fisler 2014], the author takes over 200 submissions to the Rainfall problem across five functional programming courses and manually splits the submissions based on program structure.

In [Crichton et al. 2021], the authors evaluate techniques to automate program classification using machine learning, assuming that a teacher has already provided labels indicating categories of interest. In contrast, our approach automatically discovers the labels using equivalence proofs, which can occasionally result in timeouts. It would be interesting to combine the two approaches: to use equivalence checking on a small sample to discover categories of interest, rather than manual labelling, and to then apply scalable techniques from [Crichton et al. 2021] to label entire data sets.

Ask-Elle [Gerdes et al. 2016] is an online tutor for introductory Haskell exercises. It provides feedback and incremental hints using property-based testing and strategy-based tracing. In a series of case studies, the authors demonstrate its practical value for both students and instructors. Like in our experiment, they observe different strategies in student solutions, and show how Ask-Elle benefits from having multiple reference solutions that provide strategy-specific guidance. Furthermore, they also suggest the use of student submissions as reference solutions for other submissions. However, automated custom feedback in Ask-Elle comes at a cost of writing annotations for reference solutions and manually specifying QuickCheck [Claessen and Hughes 2000] properties.

In [Geng et al. 2023], the authors propose a new approach to identify the different ways in which students interact with the grader. They extend Learn-OCaml, an online grading platform originally developed for the OCaml MOOC [Canou et al. 2017], to make it able to keep track of metrics such as grades, the number of syntax errors and the time spent on each question. They report on their experience in a functional programming course, clustering students into four fixed interaction strategies: quick-learning, hardworking, satisficing, and struggling. It would be interesting to combine the two graders and relate interaction strategies to underlying program structures.

In [Hameer and Pientka 2019], the authors report on their experience with extending Learn-OCaml for style and test quality evaluation. Like in our experiment, they also conduct their study in the context of a second year functional programming course. The authors share their insights into their Learn-OCaml extension features and the implementation, as well as their experience in using it in a course context. The paper however does not include an empirical evaluation of the integration in the course. Custom style checks for Learn-OCaml come at a cost of significant manual effort to set up a grader. Specifically, for each new exercise, the instructor has to specify dedicated syntax checks, predict unusual solutions, and write mutants to evaluate student-written tests. In contrast, we only had to provide a reference solution for each new exercise, and unit tests if desired. While the quality of student-written tests and mutation analysis are out of scope of our study, our intuition is that our resulting clusters can be used as a starting point for further mutation analysis [Prasad et al. 2024].

Program Clustering in Grading Assistants. OverCode [Glassman et al. 2015] is a grading assistant for large scale courses, providing an interactive user interface for visualizing clusters of solutions. Furthermore, OverCode offers an interface for manual manipulation of program clusters, e.g., merging the resulting clusters by adding rewrite rules. However, unlike Stainless, OverCode performs neither automated testing nor verification to check for program correctness. Complementing our approach further with OverCode's user interface and merge rules could be beneficial for programs that result in a timeout due to Stainless being overly restrictive for inner function structure.

ZEUS [Clune et al. 2020] is a grading assistant for purely functional programming. Like OverCode, ZEUS does not aim at verifying program correctness and does not provide counterexamples for incorrect programs. Like our approach, ZEUS also relies on SMT solving, but while our tool of choice uses functional induction, ZEUS uses inference rules that simulate relationships between expressions of the two programs. ZEUS is therefore more restricted with respect to recursion and introduces limitations when programs use library functions. Stainless does not have such limitations. In fact, some programs with library functions such as list concatenations appear in our resulting clusters.

Verification Tools as Grading Assistants. We notice that most past efforts in using formal techniques in a software engineering curriculum are for verification courses and not for programming courses [Noble et al. 2022]. A typical first step towards teaching undergraduate programming students how to think about program correctness is property-based testing [Earle et al. 2014; Wrenn et al. 2020]. Going one step further towards formal techniques, Dracula [Page 2005; Page et al. 2008; Vaillancourt et al. 2006] combines the ACL2 theorem prover [Kaufmann et al. 2000] with the DrScheme graphical environment [Findler et al. 2002], in introductory programming and software engineering courses. ACL2's default induction heuristic is like the functional induction in Stainless, which suggests that it would be possible to perform a similar study in ACL2, even if ACL2 is not higher-order. LEGenT [Agarwal and Karkare 2022] is a tool for personalized feedback generation, using Clara [Gulwani et al. 2018] for program clustering and the REVE [Felsing et al. 2014] equivalence checker to identify provably correct submissions. LAV [Vujosevic Janicic and Maric 2019; Vujošević-Janičić et al. 2013] is a verification tool for automated grading of imperative programming assignments. Both LEGenT and LAV are primarily targeting non-recursive programming exercises (unlike our approach, which supports recursion).

Complementary to programming assignments, some functional programming courses, including ours, also cover reasoning about programs. Recently, researchers are increasingly sharing their experience on using proof assistants for teaching [Bartzia et al. 2022]. Proof assistants are used in specialized graduate courses [Jacobsen and Villadsen 2023; Nipkow 2012; Pierce 2009], in undergraduate courses [From et al. 2022; Gambhir et al. 2023; Henz and Hobor 2011; Maxim et al. 2010; Rousselin 2023], and even high schools [Bertot et al. 2004; Guilhot 2005]. The question remains whether such proofs can be written by students in a theorem prover offering a sufficiently high-level interface, without having to learn about the proof assistant itself.

6 CONCLUSIONS

We have reported on our experience using a formal verifier for evaluation of assignments in an undergraduate functional 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, as well as to focus our attention to different types of solutions. We are thus confident that this approach represents a useful addition to automating assignment evaluation. In the future, we plan to use the approach on more exercises of the course. Furthermore, we hope to explicitly teach to students a sub-language supported by Stainless. To improve the quality of feedback reported to students, we will supply multiple reference solutions and provide more refined verification outcome summaries.

REFERENCES

Nimisha Agarwal and Amey Karkare. 2022. LEGenT: Localizing Errors and Generating Testcases for CS1. In *Proceedings of the Ninth ACM Conference on Learning @ Scale* (New York City, NY, USA) *(L@S '22)*. Association for Computing Machinery, New York, NY, USA, 102–112. https://doi.org/10.1145/3491140.3528282

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

Evmorfia Bartzia, Antoine Meyer, and Julien Narboux. 2022. Proof assistants for undergraduate mathematics and computer science education: elements of a priori analysis. In *INDRUM 2022: Fourth conference of the International Network for Didactic Research in University Mathematics*, María Trigueros (Ed.). Reinhard Hochmuth, Hanovre, Germany. https://hal.science/hal-03648357

Yves Bertot, Frédérique Guilhot, and Loic Pottier. 2004. Visualizing Geometrical Statements with GeoView. *Electr. Notes Theor. Comput. Sci.* 103 (11 2004), 49–65. https://doi.org/10.1016/j.entcs.2004.09.013

Benjamin Canou, Roberto Di Cosmo, and Grégoire Henry. 2017. Scaling up functional programming education: under the hood of the OCaml MOOC. *Proc. ACM Program. Lang.* 1, ICFP, Article 4 (aug 2017), 25 pages. https://doi.org/10.1145/3110248

Morgane Chevalier, Christian Giang, Laila El-Hamamsy, Evgeniia Bonnet, Vaios Papaspyros, Jean-Philippe Pellet, Catherine Audrin, Margarida Romero, Bernard Baumberger, and Francesco Mondada. 2022. The role of feedback and guidance as intervention methods to foster computational thinking in educational robotics learning activities for primary school. *Computers & Education* 180 (2022), 104431. https://doi.org/10.1016/j.compedu.2022.104431

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

Joshua Clune, Vijay Ramamurthy, Ruben Martins, and Umut A. Acar. 2020. Program Equivalence for Assisted Grading of Functional Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 171 (nov 2020), 29 pages. https://doi.org/10.1145/3428239

Will Crichton, Georgia Gabriela Sampaio, and Pat Hanrahan. 2021. Automating Program Structure Classification. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1177–1183. https://doi.org/10.1145/3408877.3432358

Leonardo De Moura and Nikolaj Bjørner. 2008. 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* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Clara Benac Earle, Lars-Åke Fredlund, Julio Mariño, and Thomas Arts. 2014. Teaching Students Property-Based Testing. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. 437–442. https://doi.org/10.1109/SEAA.2014.74

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 349–360. https://doi.org/10.1145/2642937.2642987

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: a programming environment for Scheme. *J. Funct. Program.* 12, 2 (mar 2002), 159–182. https://doi.org/10.1017/S0956796801004208

Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/2632320.2632346

Asta Halkjær From, Frederik Krogsdal Jacobsen, and Jørgen Villadsen. 2022. SeCaV: A Sequent Calculus Verifier in Isabelle/HOL. In *Workshop on Logical and Semantic Frameworks with Applications*. https://api.semanticscholar.org/CorpusID:248044128

Sankalp Gambhir, Simon Guilloud, Dragana Milovančević, Philipp Rümmer, and Viktor Kunčak. 2023. LISA Tool Integration and Education Plans. (2023).

Chuqin Geng, Wenwen Xu, Yingjie Xu, Brigitte Pientka, and Xujie Si. 2023. 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 V. 1 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 750–756. https://doi.org/10.1145/3545945.3569882

Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2016. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education* 27 (02 2016). https://doi.org/10.1007/s40593-015-0080-x

Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7 (March 2015), 35 pages. https://doi.org/10.1145/2699751

Benny Godlin and Ofer Strichman. 2013. Regression verification: Proving the equivalence of similar programs. *Software Testing Verification and Reliability* 23 (05 2013), 241–258. https://doi.org/10.1002/stvr.1472

Frédérique Guilhot. 2005. Formalisation en Coq et visualisation d'un cours de géométrie pour le lycée. *Technique et Science Informatiques* 24 (11 2005), 1113–1138. https://doi.org/10.3166/tsi.24.1113-1138

Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 465–480. https://doi.org/10.1145/3192366.3192387

Aliya Hameer and Brigitte Pientka. 2019. Teaching the art of functional programming using automated grading (experience report). *Proc. ACM Program. Lang.* 3, ICFP, Article 115 (jul 2019), 15 pages. https://doi.org/10.1145/3341719

Jad Hamza, Nicolas Voirol, and Viktor Kunčak. 2019. System FR: Formalized Foundations for the Stainless Verifier. *Proc. ACM Program. Lang* OOPSLA (November 2019). https://doi.org/10.1145/3360592

Martin Henz and Aquinas Hobor. 2011. Teaching Experience: Logic and Formal Methods with Coq. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 199–215.

Frederik Jacobsen and Jørgen Villadsen. 2023. On Exams with the Isabelle Proof Assistant. *Electronic Proceedings in Theoretical Computer Science* 375 (03 2023), 63–76. https://doi.org/10.4204/EPTCS.375.6

Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, USA. https://doi.org/10.1007/978-1-4615-4449-4

Angelo Kyrilov and David C. Noelle. 2015. Binary instant feedback on programming exercises can reduce student engagement and promote cheating. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '15)*. Association for Computing Machinery, New York, NY, USA, 122–126. https://doi.org/10.1145/2828959.2828968

EPFL LARA. 2023. Stainless. https://github.com/epfl-lara/stainless.

Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. 2018. Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 158 (oct 2018), 30 pages. https://doi.org/10.1145/3276528

Hendriks Maxim, Cezary Kaliszyk, Femke van Raamsdonk, and Freek Wiedijk. 2010. Teaching logic using a state-of-art proof assistant. *Acta Didactica Napocensia* 3 (06 2010).

Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Trans. Comput. Educ.* 24, 1, Article 10 (feb 2024), 43 pages. https://doi.org/10.1145/3636515

Dragana Milovančević and Viktor Kunčak. 2023. Proving and Disproving Equivalence of Functional Programming Assignments. *Proc. ACM Program. Lang.* 7, PLDI, Article 144 (jun 2023), 24 pages. https://doi.org/10.1145/3591258

Tobias Nipkow. 2012. Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–38.

James Noble, David Streader, Isaac Oscar Gariano, and Miniruwani Samarakoon. 2022. More Programming Than Programming: Teaching Formal Methods in a Software Engineering Programme. In *NASA Formal Methods*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer International Publishing, Cham, 431–450.

Martin Odersky, Lex Spoon, and Bill Venners. 2019. *Programming in Scala, Fourth Edition (A comprehensive step-by-step guide)*. Artima. https://www.artima.com/shop/programming_in_scala_4ed

Rex Page. 2005. Engineering software correctness. In *Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education* (Tallinn, Estonia) *(FDPE '05)*. Association for Computing Machinery, New York, NY, USA, 39–46. https://doi.org/10.1145/1085114.1085123

Rex Page, Carl Eastlund, and Matthias Felleisen. 2008. Functional programming and theorem proving for undergraduates: a progress report. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education* (Victoria, BC, Canada) *(FDPE '08)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/1411260.1411264

Benjamin Pierce. 2009. Lambda, the Ultimate TA Using a Proof Assistant to Teach Programming Language Foundations. 121–122. https://doi.org/10.1145/1596550.1596552

Siddhartha Prasad, Ben Greenman, Tim Nelson, and Shriram Krishnamurthi. 2024. Conceptual Mutation Testing for Student Programming Misconceptions. *CoRR* abs/2401.00021 (2024). https://doi.org/10.48550/ARXIV.2401.00021 arXiv:2401.00021

Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. 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* (Amsterdam, Netherlands) *(SPLASH Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 39–40. https://doi.org/10.1145/2984043.2989222

Pierre Rousselin. 2023. Mathematics with Coq for first-year undergraduate students. In *The Coq Workshop*.

Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LNCS, Vol. 5330)*. Springer, 274–289. https://doi.org/10.1007/978-3-540-89439-1_20

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. *SIGPLAN Not.* 48, 6 (June 2013), 15–26. https://doi.org/10.1145/2499370.2462195

E. Soloway. 1986. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (sep 1986), 850–858. https://doi.org/10.1145/6592.6594

Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and Scalable Detection of Logical Errors in Functional Programming Assignments. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 188 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360614

Dowon Song, Woosuk Lee, and Hakjoo Oh. 2021. *Context-Aware and Data-Driven Feedback Generation for Programming Assignments*. Association for Computing Machinery, New York, NY, USA, 328–340. https://doi.org/10.1145/3468264.3468598

Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability Modulo Recursive Programs. *Static Analysis*, 18. 298–315. https://doi.org/10.1007/978-3-642-23702-7_23

Dale Vaillancourt, Rex Page, and Matthias Felleisen. 2006. ACL2 in DrScheme. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications* (Seattle, Washington, USA) *(ACL2 '06)*. Association for Computing Machinery, New York, NY, USA, 107–116. https://doi.org/10.1145/1217975.1217999

Nicolas Voirol, Etienne Kneuss, and Viktor Kuncak. 2015. Counter-Example Complete Verification for Higher-Order Functions. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala* (Portland, OR, USA) *(SCALA 2015)*. Association for Computing Machinery, New York, NY, USA, 18–29. https://doi.org/10.1145/2774975.2774978

Milena Vujosevic Janicic and Filip Maric. 2019. Regression verification for automated evaluation of students programs. *Computer Science and Information Systems* 17 (01 2019), 19–19. https://doi.org/10.2298/CSIS181220019V

Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. 2013. Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments. *Inf. Softw. Technol.* 55, 6 (jun 2013), 1004–1016. https://doi.org/10.1016/j.infsof.2012.12.005

Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises. *SIGPLAN Not.* 53, 4 (jun 2018), 481–495. https://doi.org/10.1145/3296979.3192384

John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3230977.3230999

John Wrenn, Tim Nelson, and Shriram Krishnamurthi. 2020. Using Relational Problems to Teach Property-Based Testing. *The Art, Science, and Engineering of Programming* 5 (10 2020). https://doi.org/10.22152/programming-journal.org/2021/5/9