Formal Autograding in a Classroom (Extended Version)*

DRAGANA MILOVANČEVIĆ, EPFL, Switzerland and Imperial College London, UK
SAMUEL CHASSOT, EPFL, Switzerland
MARIO BUCEV, EPFL, Switzerland
MARCIN WOJNAROWSKI, EPFL, Switzerland
VIKTOR KUNČAK, EPFL, Switzerland

We report our experience in enhancing automated grading in an undergraduate programming course using formal verification. In our experiments, we deploy a program verifier to check the equivalence between student submissions and our reference solutions, alongside the existing testing-based grading infrastructure. We collect and analyse over 1700 student submissions to 11 programming exercises and show how our grader can prove submission correctness and report counterexamples. Beyond functional correctness, we were able to use the outcome of equivalence checks 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.

CCS Concepts: • **Software and its engineering → Software verification**; • **Social and professional topics → Student assessment**.

Additional Key Words and Phrases: Program Equivalence, Automated Grading, Program Clustering, Functional Induction.

ACM Reference Format:
Dragana Milovančević, Samuel Chassot, Mario Bucev, Marcin Wojnarowski, and Viktor Kunčak. 2026. Formal Autograding in a Classroom (Extended Version). In *Proceedings of ACM Transactions on Programming Languages and Systems (TOPLAS '26).* ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

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 a number of tools used in programming courses [1, 34, 47, 72, 77, 79, 85]. These tools commonly employ automated testing to assess the correctness of student submissions [53].

We observed this general trend in the functional programming course at EPFL. Whereas seven years ago the course counted around 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 throughout

*Extended version of the ESOP 2025 Experience Report Paper [58]

Authors' Contact Information: Dragana Milovančević, EPFL, Lausanne, Switzerland and Imperial College London, London, UK, dragana.milovancevic@imperial.ac.uk; Samuel Chassot, EPFL, Lausanne, Switzerland, samuel.chassot@epfl.ch; Mario Bucev, EPFL, Lausanne, Switzerland, mario.bucev@epfl.ch; Marcin Wojnarowski, EPFL, Lausanne, Switzerland, marcin.wojnarowski@epfl.ch; Viktor Kunčak, EPFL, Lausanne, Switzerland, viktor.kuncak@epfl.ch.

Manuscript submitted to ACM

the semester and for the final exam. EPFL also developed one of the first Coursera MOOCs in Europe [54], which also uses automated grading.

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 [86]. Furthermore, testing-based graders fail to provide solution-specific feedback [17, 29] on both correct and incorrect solutions. Finally, in large programming courses, the use of fully automated grading tools can lead to a situation where most students never get to speak with instructors [32], relying solely on the impersonal feedback from automated tools.

Our work aims to provide deeper and more trustworthy automated feedback, while also allowing instructors to detect and analyse solutions that would benefit from further guidance. This approach would enable high-value interactions with students when needed without significantly increasing the instructors' workload.

As an example, consider the student submissions from Figure 1, computing the `gcd` of two natural numbers. The submission from Figure 1c exhibits the same input-output behaviour as all the other submissions from Figure 1, but is notably less efficient. This submission would thus typically go unnoticed with a testing-based grader, even though it would be interesting to discuss this approach with students. 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 grader to classify submissions not only by correctness, but also based on the underlying program structure. Such motivation is also confirmed by the Rainfall study from functional programming education [25], which illustrates the variety in high-level structures of student submissions, using manual analysis. Indeed, such classification allows instructors to quickly get an idea of the different approaches followed by students, and to detect those that would benefit from additional feedback.

In this study, we propose the use of formal verification tools for automated grading and analysing the space of student submissions. We consider program equivalence proofs by functional induction as the strategy of choice for proving program correctness, and recursion patterns as the indicator of the underlying program structure. We use the Stainless verifier for Scala [37], which was previously evaluated on equivalence checking benchmarks, including programming assignments in an offline setting, translated from OCaml to Scala [57]. The theory and encouraging results of past programming languages research already suggest that verification tools can be used as grading assistants for program clustering [17, 57], but these do not include in-classroom deployment. In contrast, the goal of this paper is to report on our experience using a program verifier as a grading assistant in the *live setting* of an undergraduate programming course at EPFL. To encourage other researchers and educators to adopt 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 two experiments deploying the Stainless verifier as an automated grader for introductory exercises in our programming course. We detail our deployment, trace its evolution, present the results, and share our insights.
- We show that formal verification can reveal variations in the underlying program structure of student submissions, such as different recursion patterns. Because our approach analyses source code, it can identify differences even when submissions have identical input-output behaviour.
- We present three improvements to verifier's accuracy and performance. We show how these improvements can enhance the scalability of our grader in future deployments.
- We publish a new data set with over 1700 Scala programs, alongside the exercise material [55, 59].

```
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 this 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 patterns.

```
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 (left) and the reported feedback (right). 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.

This article is an extended version of the ESOP'25 paper entitled "Formal Autograding in a Classroom" [58]. Compared to the conference version, in this article we expand the background on equivalence checking (Section 2.2), present the evolution of the follow-up experiment (Section 5) and describe tool improvements motivated by the results of our in-course deployment (Section 6).

2 Background

One differentiating characteristic of our study is the use of a formal verifier. Our course uses Scala, so we adopt the Stainless verifier. In this section, we give background on Stainless [37] and equivalence checking in Stainless [57].

2.1 Stainless Verifier

Stainless [37, 45] is a deductive verifier for Scala [66]. 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 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 only supports 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 [37]. It relies on the Inox subsystem [46] to iteratively unfold recursive functions [80, 83] and uses SMT solvers (Z3 [21], cvc5 [3], Princess [74]) to prove or disprove those verification conditions.

Notable examples of Stainless-verified software include the Quite OK image compression algorithm [10], components of a spacecraft file system [36], a linear-time functional lexing framework [14], and various data-structures such as a purely functional map, a mutable hash table [13], as well as many functional data structures [37]. Bolts is a repository of programs verified with Stainless including most of the aforementioned programs and other smaller case studies [8].

Our approach is not specific to Stainless as such, but relies on the ability to explicitly encode inductive proofs through postconditions of recursive functions. Our deployment also exploits the ability of Stainless to generate counterexamples for incorrect programs [83], providing feedback valuable to students and instructors (Figure 2).

[1] Stainless also supports imperative programs and side effects [6, 36, 75].

2.2 Proving Program Equivalence

Our deployment makes use of a high-level functionality of Stainless to perform program equivalence checks via automated proofs by functional induction [57]. In this mode, rather than writing pre- and post-conditions, users provide specification in the form of a reference program. We use Stainless for program equivalence checking in two ways:

(1) For live autograding of individual student submissions; in this setting, pairwise equivalence checking against a reference solution can either validate submission correctness or provide a counterexample.
(2) For offline clustering analysis of student submissions; in this setting, the outcome of equivalence checking can reveal different program structures in student submissions.

In the rest of this section, we give more background on the underlying equivalence checking approach, which was introduced in [57]. Importantly, the equivalence checking is fully automated (which is necessary because the users are programming students who do not have experience with program verification) and the approach scales to larger exercises that might consist of auxiliary recursive functions.

2.2.1 Pairwise Equivalence Checking. Consider two terminating functions $f_1 : X \to Y$ and $f_2 : X \to Y$. We define the equivalence of f_1 and f_2 as: $f_1 \equiv f_2$ iff for all $x \in X$ $f_1(x) = f_2(x)$. For non-recursive functions, Stainless encodes equivalence lemmas directly as SMT solver formulas. For recursive functions, Stainless generates inductive proofs by systematically unfolding the function bodies and transforming recursive function invocations into an induction hypothesis. This approach relies on body-visible recursion, which exposes the function definitions to the verifier, and function unfolding, which transforms postconditions into assertions for recursive calls.

2.2.2 Function Call Matching. Let f_1 and f_2 be two functions that invoke auxiliary functions c_1 and c_2, respectively. Suppose that c_1 and c_2 have compatible type signatures, i.e., identical return types and identical multisets of argument types. If there exists a permutation π of the arguments such that for all $x \in X$ $c_1(x) = c_2(\pi(x))$, then calls to c_2 in f_2 can be replaced by calls to c_1, producing a transformed function f_2'. The problem of proving the equivalence of f_1 and f_2 then reduces to proving the equivalence of f_1 and f_2'. Stainless automatically generates and checks lemmas that establish (1) equivalence of auxiliary functions c_1 and c_2 under the permutation π, and (2) equivalence of the top-level functions f_1 and f_2' after substitutions. To identify candidate auxiliary function pairs (c_1, c_2) and their corresponding argument permutations, Stainless performs a *type-directed search*, followed by a *test-directed search*, which uses counterexamples to eliminate non-equivalent candidates.

We describe our autograder setup in the next section. Section 6 describes improvements to the equivalence checking mechanism that we made in response to experience from this study.

3 Formal Autograding in a Classroom

In this section, we describe our experiments 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,

Table 1. Four exercises deployed in the first 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

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 experiments took place during the fall semester of 2023 (first experiment) and the fall semester of 2024 (follow-up experiment). We prepared four autograded exercises in the first experiment (described in Table 1) and eleven autograded exercises in the follow-up experiment (described in Table 3). Each exercise is an optional individual short programming assignment to be solved and submitted on a computer.

We deployed the exercises in the last week of the course, one month before the final exam. 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. 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 ran 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 Sections 4 and 5 summarize the results.

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 [9] 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.

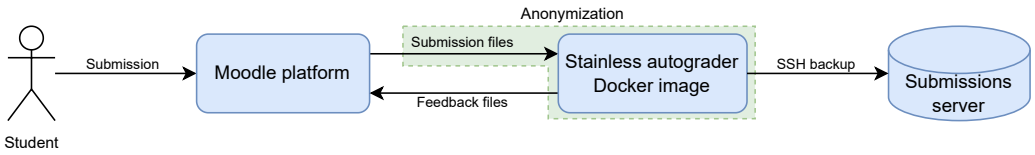


Fig. 3. Data collection pipeline.

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 unit tests for students to locally test their solutions.
(5) Write a configuration file listing the names of function(s) and 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:

- **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. Our data sets comprising 1727 Scala programs are publicly available under a permissive licence [55, 59]. We hope that the data sets will be useful to evaluate future research on programming education, which lacks public data sets.[2]

[2]In a recent systematic review [53], 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 EPFL's human research 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 first experiment. S: submissions that could not be processed due to compilation errors, or due to students submitting wrong files. TS: submissions where Stainless could not prove safety checks. I and C: submissions proven incorrect and correct, respectively. TO: 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
Total	229 (32%)	67 (9%)	219 (31%)	60 (8%)	134 (19%)		

4 First Experiment

In this section, we present and interpret the results of the first deployment of our system in fall 2023, where we deployed the four exercises described in Table 1.

4.1 Results

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

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 in 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 duplicate files, we were left with 709 submissions.[3] Around one third of the submissions are not of interest due to compilation errors (Column S), resulting in 480 syntactically valid programs.

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

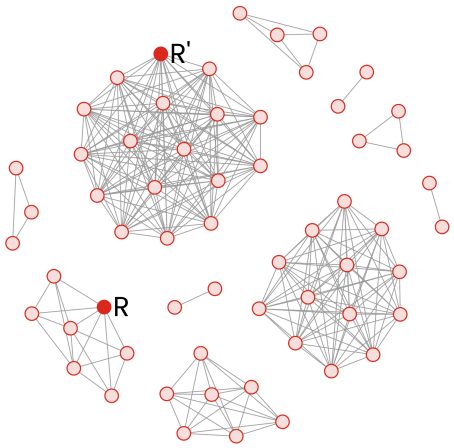
The last two columns show the results of our clustering analysis. Column "Non-Singleton" refers to the number of clusters with two or more equivalent submissions. Column "Singleton" denotes the number of submissions that are not provably equivalent to any other submission, and thus form a singleton cluster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

4.2 Lessons Learned

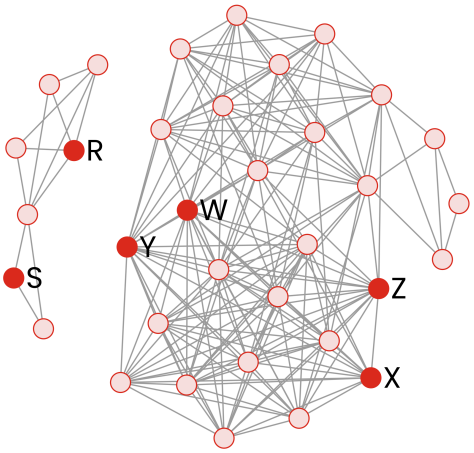
We next interpret the results of our first experiment and summarize our takeaways. Figure 4 depicts the clusters of submissions, where nodes represent submissions and edges represent direct equivalence proofs between submissions.

4.2.1 Solution Variations. We find that the verifier's success rate is lower than in the evaluation in an offline setting [57], with the majority of correct submissions timing out. We believe that this difference is due to the scarcity of reference

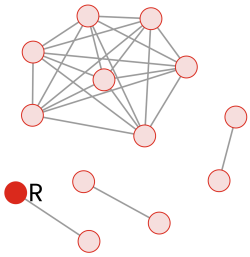
[3]It is unclear whether the duplication came from resubmissions of a student or from solution sharing between students.



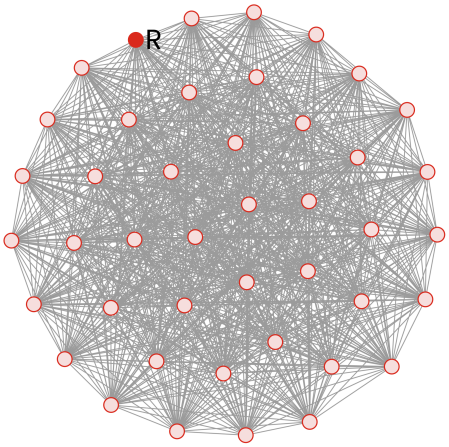
(a) Clusters of submissions for the drop exercise. The initial reference solution (node R) is in a cluster of size 7. The second reference solution (node R') is in the largest cluster.



(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 in our first experiment. Nodes represent submissions and edges represent direct equivalence proofs.

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 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 in 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 to 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 submissions.

4.2.2 Unique Solutions. Some submissions have a unique recursive structure and thus form a cluster of their own (column "Singleton" in Table 2). Singleton clusters also appear in the Rainfall study [25], where they end up in a dedicated "other/unclear composition" category. Similarly, in the related Ask-Elle studies [29], 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. 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 pattern (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 [76]. **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.

4.2.3 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.[4] Stainless could not prove this strategy 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

[4]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 [23] and RVT [31], also resulted in a timeout.

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

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.

4.2.4 Library Functions. To tame the disparity between the Scala List class and the Stainless List class[5], 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 the use of library functions [17]. Yet, support for library functions is crucial, as it allows exercising functional programming abstractions. Library functions also lead to more diversity in student submissions.

4.2.5 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 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 [84]. 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.

Programming assignments are not safety-critical on their own. However, 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 [61, 65, 81]). Namely, students are encouraged to think

[5]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 [19]).

Table 3. Eleven exercises deployed in our follow-up experiment. P: number of submissions per exercise. F and LOC: average number of functions and lines of code per submission, respectively. T: number of subtasks per exercise. R: number of reference solutions per exercise.

Name	Description	P	F	LOC	T	R
drop	Drop every n-th element from a list	93	2.6	14	1	3
gcd	Find the greatest common divisor	87	1.8	11	1	3
prime	Check if an integer is prime	184	3.6	21	1	5
infix	Implement infix boolean operators	25	8.0	18	8	1
bstbasic	Implement operations on binary search trees	62	2.1	19	2	1
context	Implement operations on contexts as ADTs	104	5.2	26	5	1
natconv	Conversion between integers and user-defined natural numbers	108	3.5	12	1	1
natop	Operations on user-defined natural numbers (+, −, *)	160	4.5	18	3	3
natparity	Implement parity checks on user-defined natural numbers	84	3.5	12	2	1
tree	Implement map, reduce, and mapReduce on binary trees	63	3.1	18	3	1
rotation	Rotate a binary search tree (left and right rotation)	48	2.1	15	2	1

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. [22].
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 Follow-Up Experiment

We conducted our follow-up experiment in fall 2024, where we deployed the same four exercises from our first experiment, alongside additional seven exercises drawn from existing course material. In this iteration of the course, the teaching material mostly remained the same. One exception is that we introduced a short lecture demonstrating program verification using Stainless.

In response to the results of our first experiment, we made the following changes to the experimental setup:

(1) We provide more reference solutions (Table 3, Column R). For the exercises from our first experiment, we use representative submissions as additional reference solutions. For the new exercises, the student assistant wrote one reference solution for each exercise and two additional reference solutions for the `natop-add` subtask.
(2) We increase the precision of exercises, both by choosing exercises with a clear structural guidance (6 out of 7 new exercises consist of several subtasks) and by writing more detailed problem statements (specifying which language constructs are allowed and which ones are not supported).
(3) We introduce a new category of feedback for suboptimal submissions (proven equivalent to suboptimal reference solutions) and provide task-level feedback[6] for larger exercises.

We present the results in Section 5.1 and then discuss the impact of these changes in Section 5.2.

5.1 Results

Table 3 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. Table 4 shows the results per exercise. Out of the 400 students

[6]Our grader was set to provide task-level feedback since the beginning, but this feature was not explored in our first experiment, because only the `infix` exercise consisted of multiple subtasks, which were all independent and trivially proven correct (no recursion, and thus no timeouts).

Table 4. Results of our follow-up experiment. S: submissions that could not be processed due to compilation errors. TS: submissions where Stainless could not prove safety checks. I and C: submissions proven incorrect and correct, respectively. TO: submissions where equivalence checks timed out. The last two columns show results of clustering analysis.

Name	S	TS	I	C	TO	Non-Singleton	Singleton
drop	24	9	25	7	28	5	13
gcd	25	4	19	25	14	2	5
prime	89	13	67	0	15	4	9
infix	2	0	2	21	0	1	0
bstbasic	10	0	21	31	0		
bstbasic-insert						1	0
bstbasic-lookup						1	0
context	24	0	18	62	0		
context-empty						1	0
context-cons						1	0
context-lookup						1	0
context-erase						1	0
context-filter						1	0
natconv	23	0	10	73	2		
natconv-toint						2	0
natop	77	0	19	18	46		
natop-add						5	3
natop-minus						1	1
natop-multiply						9	9
natparity	25	1	7	49	2		
natparity-isodd						1	2
natparity-iseven						1	2
tree	11	0	3	49	0		
tree-map						1	0
tree-reduce						1	0
tree-mapreduce						1	0
rotation	2	0	17	29	0		
rotation-left						1	0
rotation-right						1	0
Total	312 (31%)	27 (3%)	208 (20%)	364 (36%)	107 (11%)		

taking the course, 242 students agreed to participate in the study, submitting a total of 1063 submissions. After removing duplicate files, we were left with 1018 submissions. Around 31% of the submissions are not of interest due to compilation errors (Column S) which is slightly lower than in our first experiment (over 32%). Out of 706 syntactically valid programs, our grader proved correctness for over 50% (Column C). Only about 20% of syntactically valid submissions timed out, which is a significant improvement over the pilot experiment (42%). Overall, the grader was able to provide feedback fully automatically for 5 out of 11 exercises, without the need for any manual analysis (infix, bstbasic, context, tree, rotation). This result is particularly encouraging given that the majority of the exercises were from the existing course material, and not designed by us.

5.2 Discussion

We next interpret the results of our follow-up experiment and summarize our takeaways relative to our first experiment.

5.2.1 Multiple Reference Solutions. In our first experiment, we only provided one reference solution per exercise, which led to an unfortunately high rate of timeouts on correct submissions. In our follow-up experiment, we provide multiple reference solutions for 3 out of 4 exercises from our first experiment where the rate of timeouts was high (Table 3, Column R). Having a set of diverse reference solutions increased the success rate (Column TO and Column C in Table 2 vs. Table 4). For example, in our first experiment, only 3 `gcd` submissions were proven correct, while 42 submissions timed out. In our follow-up experiment, for the same `gcd` exercise, for which we provide 3 reference solutions, 25 submissions were proven correct and only 14 timed out. Out of three reference solutions for the `gcd` exercise, one uses the suboptimal algorithm from the submission in Figure 1c. One submission was proven equivalent to this reference solution, which allowed us to provide custom feedback to the student. Indeed, correct but suboptimal submissions are a great learning opportunity for students and provide additional feedback over traditional input-output testing.

In 6 out of 7 new exercises, we only provide one reference solution. At the end of the semester, clustering analysis helped us identify a few submissions that we will promote to reference solutions in future deployments. For example, in the `natconv` exercise, we identify a smaller cluster of 2 submissions that use the accumulator to implement a tail-recursive solution (as opposed to 73 non-tail recursive submissions in the larger cluster). **Takeaway:** A diverse set of reference solutions increases the success rate of the grader and can help to provide additional feedback.

5.2.2 Decomposing Exercises into Subtasks. Our approach scales to larger exercises when using appropriate decomposition, guided by problem statements or scaffolds. Most of the exercises in our follow-up experiment are composed of multiple subtasks (Table 3, Column T). The number of functions per submission (Table 3, Column F) should, in principle, be similar to the number of subtasks. Still, students are permitted to implement additional auxiliary functions, which increases the overall number of functions.

For exercises that consist of subtasks, our classification is as follows. If any subtask is incorrect or times out, the entire submission is classified as incorrect or timed-out, respectively. Conversely, a submission is classified as correct only when all subtasks are proven correct. The feedback file sent to students contains information for each subtask. For example, in the `natop` exercise, only 18/64 submissions are proven correct, due to difficulties in proving the correctness of the `natop-multiply` subtask. However, in the `natop-add` subtask, 50/64 submissions are proven correct and in the `natop-minus` subtask, 63/64 submissions are proven correct. **Takeaway:** Decomposing exercises into subtasks helps to make our approach scalable. A clear interface can help guide equivalence checking and provide fine-grained feedback, while scaling to larger, more interesting and practical exercises.

5.2.3 Early Exposure to Formal Methods. The primary goal of our experiments was to evaluate opportunities for fully automated deployment of formal verification tools in courses that teach functional programming and software construction, even if students are not familiar with the verification tool used for equivalence checking. That said, a few weeks prior to our follow-up experiment, students were exposed to additional information about Stainless and formal methods, which was not presented in the previous course iteration. The reason we introduced formal methods concepts was because we felt that they are a core topic of the course on software construction. We explained the basic principles on which Stainless operates, the meaning of **require** and **ensuring** clauses, as well as class invariants in the context of verification tools. Moreover, students solved a homework exercise that involved finding invariants and using Stainless to prove correctness of simple programs.

```
def treeMapReduce(tree: IntTree, mapper: Int => String, reducer: (String, String) => String): String = {
  tree match
    case Leaf(value) => mapper(value)
    case Branch(left, right) => reducer(treeMapReduce(left, mapper, reducer),treeMapReduce(right, mapper, reducer))
} ensuring (res => res == reduce(treeMapString(tree, mapper), reducer))

def treeMapString(tree: IntTree, mapper: Int => String): StrTree = tree match
  case Leaf(value) => StrTree.Leaf(mapper(value))
  case Branch(left, right) => StrTree.Branch(treeMapString(left, mapper), treeMapString(right, mapper))

def reduce(mappedTree: StrTree, reducer: (String, String) => String): String = mappedTree match
  case StrTree.Leaf(value) => value
  case StrTree.Branch(left, right) => reducer(reduce(left, reducer), reduce(right, reducer))
```

Fig. 6. MapReduce proof written by a student

After this additional exposure to formal methods and Stainless, in our follow-up experiment, we found that some students wrote additional specifications in their submissions, which were not needed for solving the autograded exercises. For example, Figure 6 shows a part of one submission to the tree–mapreduce task, where the student wrote a specification of their treeMapReduce function. The property was proven by Stainless, which presumably helped the student gain confidence before submitting their solution for equivalence checking. Moreover, students occasionally wrote preconditions on their auxiliary functions, to prevent safety errors that would only arise for inputs that are not of interest in the context of the outer functions. Consequently, the knowledge of Stainless enabled students to write code more naturally and to benefit from verification when solving exercises. **Takeaway:** We found that teaching formal methods was synergistic with, even if not necessary for, deploying a formal autograder in a software construction course.

5.2.4 Immediate vs. Delayed Feedback. Because students have local access to Stainless, they have access to immediate feedback on compilation errors and safety errors. However, we limit the output of equivalence checking to be only available upon submitting to Moodle (typically within five minutes). This deferral is in line with 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 [15]. For now, our deployment implies a setting which overall provides three levels of feedback: (1) instantaneous local feedback from the compiler, safety checker, and the test suite (2) near-instantaneous feedback from the equivalence checker upon assignment submission and (3) additional cumulative feedback of clustering analysis for the entire class. We found this setting an appropriate, satisfying balance between automation and manual instructor interaction. We however do not measure the effect of such layered feedback on learning in this study. A recent review paper suggests that this setting is largely unexplored and would benefit from further education research [53]. **Takeaway:** Providing direct access to verifier used in autograding can provide additional modes of feedback to students; the impact of such feedback would be interesting to study in the future.

6 Using Experience to Drive Tool Improvements

Throughout our experiments, by evaluating equivalence checking on 1727 student submissions, we were able to identify and propose several improvements in the Stainless verifier. In this section, we first briefly comment on two recent

features whose development was influenced by our autograding experiments: termination measure transfer and support for mutual recursion. We then present our development of subtask caching, and show how it improves the scalability of equivalence checking.

6.1 Termination Measure Transfer

Termination checking is integrated in Stainless, like in many other verifiers and proof assistants, by means of synthesizing and verifying termination measures. Prior to our autograding experiments, our intuition was that equivalent programs written by different authors would require different termination proofs, which would correspond to different termination measures in Stainless. Our experience shows otherwise: equivalent programs, even when written by different students, often terminate for the same reason (evidenced by the same termination measure). This discovery led us to developing measure transfer in Stainless, a simple instance of proof transfer for termination proving, which scales to a large number of equivalent programs, even for complex termination measures. The measure transfer algorithm and its evaluation on programming exercises and program refactoring are presented in detail in [60].

Our grader already reports additional feedback on the outcome of termination checking (Section 3, Paragraph Feedback Generation), using built-in measure inference in Stainless. To benefit from the additional scalability of measure transfer, the instructors can optionally provide termination measures in the reference solutions.

6.2 Function Matching and Mutual Recursion

In our follow-up experiment, we encountered several submissions with mutually recursive functions. During the clustering analysis, it became apparent that the implementation of function call matching in Stainless did not support mutual recursion (Section 2.2). Upon this observation, we modified the function matching algorithm to also match mutually recursive functions. This modification enabled us to prove the equivalence of mutually recursive `isOdd` and `isEven` functions, which appear in 5 submissions in the `natparity` exercise.

6.3 Caching of Subtasks

When exercises consist of subtasks that build on one another, our grader runs separate Stainless instances for each subtask, and provides task-level feedback to students (Section 5.2.2). In principle, the grader should be able to verify each subtask independently. However, because the grader does not reuse information from preceding runs, it will repetitively regenerate and check equivalence sublemmas for each subtask. Furthermore, when writing reference solutions for the final subtask, one would have to write multiple copies of each solution to capture all the combinations of smaller subtasks. The number of reference solutions could thus quickly explode as a function of the number of subtasks.

To address both issues, we extend the equivalence checking algorithm in Stainless to use subtask caching for equivalent functions. In our clustering analysis, upon checking a subtask, the cache will store all the equivalent top-level functions for that subtask. Later, during the equivalence checking for subsequent subtasks, if the function matching algorithm encounters an equivalence sublemma whose both sides already appear in the cache, the algorithm can trivially bypass this sublemma. In the cache, we additionally store reference solutions to subtasks. Because the grader already relies on the correctness of all reference solutions, we can take the equivalence of reference solutions as an axiom.

For example, in the `natop` exercise, the `natop–multiply` subtask has a single reference solution, `multiplyR`, which invokes `addR`, one of the 3 reference solutions to the `natop–add` subtask. When checking the equivalence of `multiply` function in student submissions and the `multiplyR` in the reference solution, the function matching would attempt to prove the equivalence of the `add` function in the student submission and `addR`. In several student submissions, the equivalence

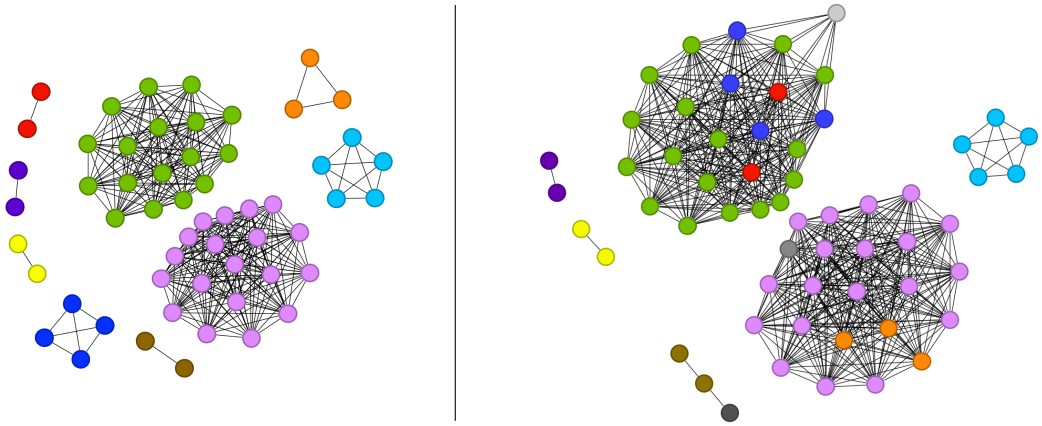


Fig. 7. Non-singleton clusters of submissions for the natop–multiply subtask, when using Stainless without the subtask cache (left) and after adding the subtask cache (right). When using the cache, 3 small clusters (coloured in red, blue and orange) got merged with one of the 2 largest clusters (coloured in green and pink). The 3 grey nodes on the right picture are not present on the left picture because those submissions were initially not proven equivalent to any other submission (singleton clusters).

proof for multiply and multiplyR would time out because the add function could not be proven equivalent to addR. However, in several of those submissions, the add function was instead proven equivalent to either addRn or addRnm. When using the subtask caching, by having all of addR, addRn, addRnm and add in the cache, our system was able to complete the equivalence proof for multiply and multiplyR. Overall, in natop exercise, caching increased the number of submissions proven correct from 18 to 22 (Table 4, Column C).

To further illustrate the impact of this feature, we run the clustering analysis for the natop exercise using subtask caching. Figure 7 illustrates the natop–multiply clusters before and after implementing the subtask cache. Caching the results of the natop–add subtask reduces the number of non-singleton clusters in the natop–multiply subtask from 9 to 6 (Table 4, Colum Non-Singleton). Similarly, the number of singleton clusters drops from 9 to 6 (Table 4, Column Singleton). The reduction in number of clusters implies fewer manual checks (typically, one per cluster). At the same time, this reduction does not result in any loss of information about solution variations. Although subtask caching merges clusters that were previously split due to variations in the natop–add implementation, those variations remain visible in the clustering analysis of the natop–add subtask. Submission clustering when using subtask caching thus mimics the behaviour of hierarchical clustering [42], where collapsing distinctions at an earlier level leads to fewer clusters at subsequent levels while still preserving those distinctions within the lower-level analyses.

We are confident that tool improvements presented here will improve the scalability of our grader in the future deployments, as we continue to use our approach on larger exercises.

7 Limitations, Threats to Validity, and Future Work

In this section, we consider some limitations of our approach and discuss avenues for future work.

Course Size. Our analysis only uses data from 1727 submissions from one course at one university, and does not necessarily represent the general trend in programming education. We found this setting (over 200 students, 11 exercises) to be a reasonable size for our experiments, which enabled us to identify interesting aspects of in-class deployment.

Size of the Exercises. In our experiments, we consider small introductory exercises, with up to 62 LOC and 16 LOC on average. Future research could empirically test the boundaries of this approach on larger exercises.

Programming Language. While our study uses the Scala language and the Stainless verifier, our approach also applies to other languages and verifiers. In particular, equivalence proofs by functional induction are applicable to recursive programs in general [52]. Notable examples include functional induction in Rocq [40] and Lean [18], recursion induction [64] or computation induction [63] in Isabelle, default induction heuristic in ACL2 [43]. Furthermore, tools such as REVE [23], RVT [31] or SymDiff [44] have shown that equivalence checking is also feasible for imperative programs, beyond the examples that we presented in our study.

Checking Syntax and Style. While it would be possible to further incorporate techniques for syntax and style checking [35, 38], in this report, we focus on semantic program structure defined by recursion patterns. 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, clustering by recursion patterns in some cases does not differentiate purely functional programs from programs with loops and mutations. The reason is that Stainless automatically transforms loops into tail-recursive functions during verification [7]. For example, submission gcdY with a while loop and variable mutation[7] is proven equivalent to purely functional submission gcdW, and thus in the same cluster (Figure 1b). To address those limitations, future research could consider a more fine-grained clustering to differentiate recursion and loops, by performing syntax checks complementary to inductive equivalence proofs. Similar checks are already used by MOOC graders [35, 54].

Resource Use Properties. Our notion of equivalence does not account for resource usage (such as running time or memory use) directly, but only indirectly, as programs that have different complexity often use different recursion patterns. Explicitly accounting for resources could be done using source-to-source instrumentation, and was shown feasible in Leon [48, 49], a system similar to Stainless.

Dependence of Clustering on Equivalence Prover. When performing equivalence checks on two programs, our approach can yield three possible outcomes: (1) an equivalence proof, (2) a counterexample, or (3) a timeout. The first two outcomes have a clear declarative specification. Because we focus on provable equivalence, we classify submissions for which equivalence checks time out into different clusters. Consequently, the clusters we obtain are affected by the capabilities of the equivalence prover. It would be useful to have declarative descriptions of cases where the prover is expected to succeed or fail. Past work on program schemes could be a possible starting point for such descriptions [2, 67].

In-Person Assessments. In our experiments, we only consider optional weekly exercises, but our approach also applies to exams and in-person assessments. At EPFL, our course currently uses a testing-based autograding infrastructure for the final exam. Our Stainless-backed autograder is built on top of the same infrastructure, and can run as is in the exam setting. More broadly, many universities are rapidly moving towards computer-based testing facilities, where students

[7] 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.

take in-person exams on a computer [50, 88]. We hope that our experiments will encourage future adoption of formal autograding for in-person assesments, including in computer-based testing facilities.

Using Large Language Models. Beyond the scope of this paper is the use of large language models (LLMs) in the context of teaching and evaluating programming skills. Our deployment assumes that students do not use LLMs for solving exercises. We believe it is important for students to understand the principles of functional programming by solving these small exercises by hand, much like students at some point in their education learn to solve math exercises without a calculator. Our approach therefore remains valuable for these introductory exercises despite advances in LLMs. To ensure that the students comply with the no LLM policy, our system could be deployed in a controlled on-campus environment, such as a computer-based testing facility.

Orthogonally to our work, researchers have proposed new data sets for LLM-generation of verified programs [11]. The resulting open data sets from our experiments [55, 56, 59] could also be used to evaluate the nature of LLM-generated solutions and compare them to student solutions. Finally, LLMs could be used, in principle, to evaluate student solutions and provide natural language feedback to students [70]. Such use of LLMs as judges is complementary to our use of formal verification, as LLMs alone cannot be trusted to rule out subtle errors (such as the ones arising for large or edge-case inputs).

Structural Diversity vs. Structural Conformity. Our work celebrates the diversity of student submissions, and encourages instructors to provide multiple reference solutions. However, at the introductory level, some instructors instead expect that every student follows the same canonical template, exercising a specific structural pattern. Our approach also applies to that setting, where a single reference solution suffices for the equivalence checking, and the expected outcome is that all student submissions belong to the same cluster. For example, in our follow-up experiment, for the three exercises that operate on trees (bstbasic, tree, rotation), all correct student submissions follow the same recursion pattern, and are all proven equivalent to a single reference solution.

Educational Impact. Our study illustrates how feedback from program verifiers can enhance automated grading, but we do not measure the impact of this feedback on learning outcomes. Future research on programming education could go one step further and investigate the educational impact, by conducting a survey filled out by students or a more quantitative analysis via controlled studies.

8 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.

8.1 Functional Programming Education

The Rainfall problem [78], originally studied in the field of imperative programming education, has recently become an insightful benchmark in functional programming education. In [25], 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 [20]. In contrast, our approach automatically discovers clusters of submissions using equivalence proofs.

Learn-OCaml is an online grading platform for the OCaml MOOC [12]. Researchers have proposed extensions to the platform allowing assessment of style and test quality [35], as well as understanding how students interact with the grader [28]. 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 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 recursion patterns.

Ask-Elle [29] is an online tutor for introductory Haskell exercises, providing feedback and incremental hints using property-based testing and strategy-based tracing. Like in our experiments, 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 [16] 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 unit tests, leaving time for instructors to discuss specific solutions with students.

8.2 Program Clustering in Grading Assistants

The idea of using program equivalence to detect algorithmic similarity was previously explored in the ZEUS grading assistant [17]. 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 programs with library functions. While both [17] and [57] focus on empirical evaluation of equivalence checking, the focus of our analysis is on *understanding* the resulting clusters and corresponding recursion patterns.

OverCode [30] is a grading assistant for large scale courses, providing an interactive user interface for visualizing clusters of student submissions. 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 discovered by Stainless, in particular for incorrect solutions to identify patterns of student misunderstandings.

8.3 Verification Tools as Grading Assistants

LEGenT [1] is a tool for personalized feedback generation, using Clara [34] for program clustering and REVE [23] to identify provably correct submissions. LAV [84] 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 [68, 69, 82] combines the ACL2 theorem prover [43] with the DrScheme graphical interface [24], 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 experiments. The main difference is that, in their setting, the goal is to teach 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 language does not support higher-order functions.

Recently, researchers are increasingly sharing their experience on using proof assistants for teaching [4], both for mathematics and computer science programs. Proof assistants have been increasingly finding their way in specialized graduate courses [41, 62, 71], in undergraduate courses [26, 27, 39, 51, 73], and even high schools [5, 33]. 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.

9 Conclusions

We have reported our experience in using a formal verifier to evaluate student submissions 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 direct our attention towards 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 plan to progressively use our approach on larger assignments. We have seen that modular equivalence checkers, such as Stainless, naturally scale to larger assignments when they consist of smaller subtasks. 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. Beyond introductory programming courses, in the long run, we will also consider metrics to explicitly cluster solutions by algorithmic complexity. Such feature would be particularly desirable in competitive programming and algorithms courses [76, 87].

In the long run, a particularly exciting avenue is the creation of a repository of trusted and verified programs, such as those that operate on functional data structures. Such repository would serve as a reusable library of verified reference solutions, shared across courses and institutions, and would be of general use both in and out of classroom.

Acknowledgments

We thank the anonymous TOPLAS reviewers and ESOP 2025 reviewers for their thoughtful feedback and suggestions, and thank Sophia Drossopoulou for many helpful comments. This project was supported in part by the DRIL funds from EPFL (Digital Resources for Instruction and Learning), by the UKRI Future Leaders Fellowship MR/V024299/1, and by the Swiss National Science Foundation Project number 200021_219474. We thank Martin Odersky and Patrick Jermann for their feedback and help with our DRIL project application, which contains the prototype of our deployment. We thank Tanja Käser and Patrick Jermann for their help and guidance with obtaining the permissions from EPFL's Human Research Ethics Committee to publish the data and the results of our experiments. We thank Léa Blancher for preparing the exercises in our follow-up experiment. We thank EPFL students of the Software Construction course in fall 2023 and fall 2024 for contributing their submissions and making this case study possible.

Availability of Data and Software

Stainless verifier is under active development and is available at [45]. The complete data sets from this paper are available in open access Zenodo repositories [55, 59].

References

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

[2] Edward Ashcroft, Zohar Manna, and Amir Pnueli. 1973. Decidable Properties of Monadic Functional Schemas. *J. ACM* 20, 3 (July 1973), 489–499. doi:10.1145/321765.321780

[3] 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*. 415–442. doi:10.1007/978-3-030-99524-9_24

[4] 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, HAL, Hanovre, Germany. https://hal.science/hal-03648357

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

[6] Régis William Blanc. 2017. Verification by Reduction to Functional Programs. (2017), 191. doi:10.5075/epfl-thesis-7636

[7] Régis William Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. 2013. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Scala Workshop*. doi:10.1145/2489837.2489838

[8] Bolts 2026. epfl-lara/bolts. https://github.com/epfl-lara/bolts. https://github.com/epfl-lara/bolts original-date: 2017-10-12T10:48:10Z.

[9] Matt Bovel and Hamza Remmal. 2024. An Automatic Grading System for Large Classes. In *Proceedings of the 52nd Annual Conference of SEFI, Lausanne, Switzerland*. doi:10.5281/zenodo.14256923

[10] Mario Bucev and Viktor Kunčak. 2022. Formally Verified Quite OK Image Format. In *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design–FMCAD 2022*. TU Wien, 343–348.

[11] Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. 2025. A benchmark for vericoding: formally verified program synthesis. arXiv:2509.22908 [cs.SE] https://arxiv.org/abs/2509.22908

[12] 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 (2017), 25 pages. doi:10.1145/3110248

[13] Samuel Chassot and Viktor Kuncak. 2024. Verifying a Realistic Mutable Hash Table. *CoRR* abs/2107.08824 (2024). arXiv:2107.08824 https://arxiv.org/abs/2107.08824

[14] Samuel Chassot and Kunčak. 2026. Formally Verified Linear-Time Invertible Lexing. In *Proceedings of the 38th International Conference on Computer Aided Verification* (Lisbon, Portugal) *(CAV'26)*.

[15] 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. doi:10.1016/j.compedu.2022.104431

[16] 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)*. 268–279. doi:10.1145/351240.351266

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

[18] Lean Community. 2025. Functional Induction in Lean. https://leanprover-community.github.io/mathlib4_docs/Lean/Meta/Tactic/FunInd.html.

[19] Scala Community. 2025. Variance in Scala. https://docs.scala-lang.org/tour/variances.html.

[20] 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 (SIGCSE '21)*. 1177–1183. doi:10.1145/3408877.3432358

[21] 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 (TACAS'08/ETAPS'08)*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[22] Brijesh Dongol, Catherine Dubois, Stefan Hallerstede, Eric Hehner, Carroll Morgan, Peter Müller, Leila Ribeiro, Alexandra Silva, Graeme Smith, and Erik de Vink. 2024. On Formal Methods Thinking in Computer Science Education. *Form. Asp. Comput.* (2024). doi:10.1145/3670419

[23] 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 (ASE '14)*. 349–360. doi:10.1145/2642937.2642987

[24] 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.* (2002), 159–182. doi:10.1017/S0956796801004208

[25] Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER'14)*. 35–42. doi:10.1145/2632320.2632346

[26] Asta Halkjær From, Frederik Krogsdal Jacobsen, and Jørgen Villadsen. 2022. 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)*. 38–55. doi:10.4204/EPTCS.357.4

[27] Sankalp Gambhir, Simon Guilloud, Dragana Milovančević, Philipp Rümmer, and Viktor Kunčak. 2023. LISA Tool Integration and Education Plans.

[28] 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 (SIGCSE 2023)*. 750–756. doi:10.1145/3545945.3569882

[29] 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* (2016). doi:10.1007/s40593-015-0080-x

[30] 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.*, Article 7 (2015), 35 pages. doi:10.1145/2699751

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

[32] William G. Griswold. 2024. 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 (SIGCSE 2024)*. 415–421. doi:10.1145/3626252.3630844

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

[34] 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 (PLDI 2018)*. 465–480. doi:10.1145/3192366.3192387

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

[36] Jad Hamza, Simon Felix, Viktor Kunčak, Ivo Nussbaumer, and Filip Schramka. 2022. From Verified Scala to STIX File System Embedded Code Using Stainless. In *NASA Formal Methods*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer International Publishing, Cham, 393–410. doi:10.1007/978-3-031-06773-0_21

[37] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. 2019. System FR: Formalized Foundations for the Stainless Verifier. *Proc. ACM Program. Lang.*, Article 166, 30 pages. doi:10.1145/3360592

[38] Rowan Hart, Brian Hays, Connor McMillin, El Kindi Rezig, Gustavo Rodriguez-Rivera, and Jeffrey A. Turkstra. 2023. 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 (SIGCSE 2023)*. 799–805. doi:10.1145/3545945.3569817

[39] 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. doi:10.1007/978-3-642-25379-9_16

[40] INRIA. 2021. Functional Induction in Coq. https://coq.inria.fr/refman/using/libraries/funind.html.

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

[42] Joe H. Ward Jr. 1963. Hierarchical Grouping to Optimize an Objective Function. *J. Amer. Statist. Assoc.* 58, 301 (1963), 236–244. doi:10.1080/01621459.1963.10500845

[43] 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

[44] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the 24th International Conference on Computer Aided Verification* (Berkeley, CA) *(CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 712–717. doi:10.1007/978-3-642-31424-7_54

[45] LARA. 2023. Stainless. https://github.com/epfl-lara/stainless.

[46] EPFL LARA. 2025. Stainless. https://github.com/epfl-lara/inox/.

[47] 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.*, Article 158, 30 pages. doi:10.1145/3276528

[48] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based resource verification for higher-order functions with memoization. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 330–343. doi:10.1145/3009837.3009874

[49] Ravichandhran Madhavan and Viktor Kuncak. 2014. Symbolic Resource Bound Inference for Functional Programs. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 762–778. doi:10.1007/978-3-319-08867-9_51

[50] Anastasiya Markova, Anish Kasam, Bryce Hackel, Marina Langlois, and Sam Lau. 2026. Designing and Implementing Skill Tests at Scale: Frequent, Computer-Based, Proctored Assessments with Minimal Infrastructure Requirements. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1* (USA) *(SIGCSE TS 2026)*. Association for Computing Machinery, New York, NY, USA, 701–707. doi:10.1145/3770762.3772518

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

[52] John McCarthy. 1963. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 35. Elsevier, 33–70. doi:10.1016/S0049-237X(08)72018-4

[53] 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 (2024), 43 pages. doi:10.1145/3636515

[54] Heather Miller, Philipp Haller, Lukas Rytz, and Martin Odersky. 2014. Functional programming for all! scaling a MOOC for students and professionals alike. In *ICSE Companion*. ACM, 256–263.

[55] Dragana Milovancevic, Samuel Chassot, Mario Bucev, Marcin Wojnarowski, and Viktor Kunčak. 2025. *Formal Autograding in a Classroom (Extended Version Artifact)*. doi:10.5281/zenodo.17600207

[56] Dragana Milovancevic and Viktor Kuncak. 2023. *Proving and Disproving Equivalence of Functional Programming Assignments (Artifact)*. doi:10.5281/zenodo.7810840

[57] Dragana Milovančević and Viktor Kunčak. 2023. Proving and Disproving Equivalence of Functional Programming Assignments. *Proc. ACM Program. Lang.*, Article 144, 24 pages. doi:10.1145/3591258

[58] Dragana Milovančević, Mario Bucev, Marcin Wojnarowski, Samuel Chassot, and Viktor Kunčak. 2025. Formal Autograding in a Classroom. In *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*. https://infoscience.epfl.ch/handle/20.500.14299/205841.2

[59] Dragana Milovančević, Mario Bucev, Marcin Wojnarowski, Samuel Chassot, and Viktor Kunčak. 2025. *Formal Autograding in a Classroom (Artifact)*. doi:10.5281/zenodo.14624668

[60] Dragana Milovančević, Carsten Fuhs, Mario Bucev, and Viktor Kunčak. 2024. Proving Termination via Measure Transfer in Equivalence Checking. In *International Conference on Integrated Formal Methods (iFM)*.

[61] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 252–269. doi:10.1145/3132747.3132748

[62] 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.

[63] Tobias Nipkow. 2022. Programming and Proving in Isabelle/HOL. https://isabelle.in.tum.de/dist/Isabelle2022/doc/prog-prove.pdf.

[64] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media. https://doi.org/10.1007/3-540-45949-9

[65] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 530–545. doi:10.1145/3445814.3446748

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

[67] Luke Ong. 2015. Higher-Order Model Checking: An Overview. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. IEEE Computer Society, 1–15. doi:10.1109/LICS.2015.9

[68] Rex Page. 2005. Engineering software correctness. In *Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education (FDPE'05)*. 39–46. doi:10.1145/1085114.1085123

[69] 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 (FDPE'08)*. 21–30. doi:10.1145/1411260.1411264

[70] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. In *Proceedings of the 16th International Conference on Educational Data Mining*. International Educational Data Mining Society, Bengaluru, India, 370–377. doi:10.5281/zenodo.8115653

[71] Benjamin C. Pierce. 2009. 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 (ICFP '09)*. 121–122. doi:10.1145/1596550.1596552

[72] 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 (SPLASH Companion 2016)*. 39–40. doi:10.1145/2984043.2989222

[73] Pierre Rousselin. 2023. Mathematics with Coq for first-year undergraduate students.

[74] Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*. 274–289. doi:10.1007/978-3-540-89439-1_20

[75] Georg Stefan Schmid and Viktor Kuncak. 2022. Generalized Arrays for Stainless Frames. In *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)*, Bernd Finkbeiner and Thomas Wies (Eds.). Springer, 332–354. doi:10.1007/978-3-030-94583-1_17

[76] Anjali Singh, Anna Fariha, Christopher Brooks, Gustavo Soares, Austin Z. Henley, Ashish Tiwari, Chethan M, Heeryung Choi, and Sumit Gulwani. 2024. Investigating Student Mistakes in Introductory Data Science Programming. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)*. 1258–1264. doi:10.1145/3626252.3630884

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

[78] E. Soloway. 1986. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM* (1986), 850–858. doi:10.1145/6592.6594

[79] Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and Scalable Detection of Logical Errors in Functional Programming Assignments. *Proc. ACM Program. Lang.*, Article 188, 30 pages. doi:10.1145/3360614

[80] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability Modulo Recursive Programs. In *Static Analysis*, Eran Yahav (Ed.). 298–315. doi:10.1007/978-3-642-23702-7_23

[81] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2022. Giallar: push-button verification for the qiskit Quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 641–656. doi:10.1145/3519939.3523431

[82] 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 (ACL2'06)*. 107–116. doi:10.1145/1217975.1217999

[83] 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 (SCALA 2015)*. 18–29. doi:10.1145/2774975.2774978

[84] 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.* (2013), 1004–1016. doi:10.1016/j.infsof.2012.12.005

[85] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises. 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, 481–495. doi:10.1145/3192366.3192384

[86] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER'18)*. Association for Computing Machinery, 51–59. doi:10.1145/3230977.3230999

[87] Jialu Zhang, De Li, John Charles Kolesar, Hanyuan Shi, and Ruzica Piskac. 2023. Automated Feedback Generation for Competition-Level Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi:10.1145/3551349.3560425

[88] Craig Zilles, Matthew West, Geoffrey Herman, and Timothy Bretl. 2019. Every University Should Have a Computer-Based Testing Facility. 414–420. doi:10.5220/0007753304140420

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009