Formal Autograding in a Classroom (Experience Report)

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

May 07, 2025

Manual Grading is Difficult

Functional Programming @ EPFL

Example Exercise

Implement a function that takes a list of integers ls and an integer n, and returns the list obtained by dropping every n-th element from ls. For example, the input (List(1, 2, 3, 4, 5, 6, 7, 8, 9), 3) should lead to the output List(1, 2, 4, 5, 7, 8).

Example Exercise

Implement a function that takes a list of integers ls and an integer n, and returns the list obtained by dropping every n-th element from ls. For example, the input (List(1, 2, 3, 4, 5, 6, 7, 8, 9), 3) should lead to the output List(1, 2, 4, 5, 7, 8).

```
def drop(ls: List[Int], n: Int): List[Int] =
  require(n >= 0)
  def loop(ls: List[Int], i: Int): List[Int] =
    require(i > 0 && n > 0)
    ls match
      case x :: xs =>
        if i == 1 then
          loop(xs, n)
        else
          x :: loop(xs, i - 1)
      case Nil => Nil
  if n == 0 then ls else loop(ls, n)
```

Example Exercise

Implement a function that takes a list of integers ls and an integer n, and returns the list obtained by dropping every n-th element from ls. For example, the input (List(1, 2, 3, 4, 5, 6, 7, 8, 9), 3) should lead to the output List(1, 2, 4, 5, 7, 8).

```
def drop(ls: List[Int], n: Int): List[Int] =
  require(n >= 0)
  def loop(ls: List[Int], i: Int): List[Int] =
    require(i > 0 && n > 0)
    ls match
      case x :: xs =>
        if i == 1 then
          loop(xs, n)
        else
          x :: loop(xs, i - 1)
      case Nil => Nil
  if n == 0 then ls else loop(ls, n)
```

Is this solution correct? ✓

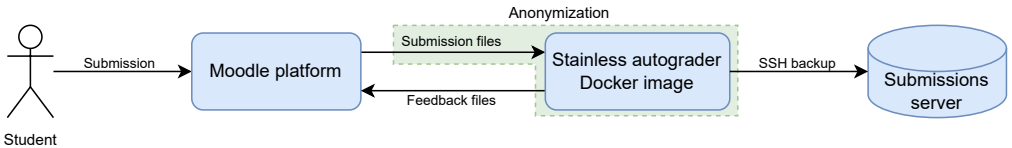
How Does This Scale?

- FP @ ETF Belgrade in 2018: 30 students
- FP @ EPFL Lausanne in 2019: 200 students
- FP @ EPFL Lausanne in 2023: 400 students
  - 11 exercise sessions
  - 9 homework projects
  - midterm and final exam
- Coursera Scala MOOC: over 200k students

Formal Autograding

- So far, in our course, testing-based automated grading
  - Inaccurate, flawed assessment at scale
  - Impersonal, fails to provide solution-specific feedback
- Our experiment: formal automated grading via program equivalence proving
  - Formal techniques as a guarantee that programs are never wrongly classified
  - Like human graders, examine the source code instead of just running it

Our Grader


Anonymization
Submission
Moodle platform
Submission files
Stainless autograder
Docker image
Feedback files
SSH backup
Submissions
server
Student

Formal Autograding backed by the Stainless verifier

stainless.epfl.ch

- Open-source deductive verifier for Scala programs
- Used to verify data structures, blockchain clients, compression and other algorithms
- Bolts (Stainless Verified Scala): github.com/epfl-lara/bolts
- ASPLOS'22 Tutorial: epfl-lara.github.io/asplos2022tutorial

Internally: Program Equivalence Checking

- Formal verification is difficult!
  - Case studies show ratios such as 9 lines of specifications per executable line[1]
- Solution: Program equivalence checking as push-button verification
  - Specification is an executable program, instead of pre- and post-conditions
  - Suitable for non-expert users

[1] Mario Bucev and Viktor Kunčak. Formally verified quite ok image format. FMCAD'22.

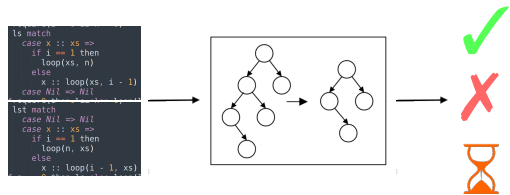
Background: Pairwise Equivalence Checking in Stainless

Definition of program equivalence?

- Given: candidate program f: $X \to Y$, proven terminating, and reference program m: $X \to Y$, proven terminating. A candidate program f is equivalent to a reference program m iff $f(x) = m(x)$ for all $x \in X$.

Main techniques:[1]

- proofs by (functional) induction
- function call matching
- clustering algorithm



[1] Dragana Milovančević and Viktor Kunčak. Proving and Disproving Equivalence of Functional Programming Assignments. PLDI'23.

Background: Functional Induction in Stainless

```
def dropM(ls: List[Int], n: Int) =
  require(n >= 0)
  def loop(ls: List[Int], i: Int) =
    require(i > 0 && n > 0)
    ls match
      case x :: xs =>
        if i == 1 then
          loop(xs, n)
        else
          x :: loop(xs, i - 1)
      case Nil => Nil
  if n == 0 then ls
  else loop(ls, n)
```

```
def dropF(ls: List[Int], n: Int) =
  require(n >= 0)
  def rec(actual: Int, b: Int, ls: List[Int]) =
    require((actual >= 1) && (b > 0))
    ls match
      case Nil => Nil
      case first :: next =>
        if (actual == 1) then
          rec(b, b, next)
        else
          first :: rec(actual - 1, b, next)
  if (n == 0) then ls
  else rec(n, n, ls)
```

Background: Functional Induction in Stainless

```
def dropM(ls: List[Int], n: Int) =
  require(n >= 0)
  def loop(ls: List[Int], i: Int) =
    require(i > 0 && n > 0)
    ls match
      case x :: xs =>
        if i == 1 then
          loop(xs, n)
        else
          x :: loop(xs, i - 1)
      case Nil => Nil
  if n == 0 then ls
  else loop(ls, n)
```

```
def dropF(ls: List[Int], n: Int) =
  require(n >= 0)
  def rec(actual: Int, b: Int, ls: List[Int]) =
    require((actual >= 1) && (b > 0))
    ls match
      case Nil => Nil
      case first :: next =>
        if (actual == 1) then
          rec(b, b, next)
        else
          first :: rec(actual - 1, b, next)
  if (n == 0) then ls
  else rec(n, n, ls)
```

- Inner function matching using type-directed search

Background: Functional Induction in Stainless

```
def loop(ls: List[Int], i: Int) = {
  require(i > 0 && n > 0)
  ls match
    case x :: xs =>
      if i == 1 then
        loop(xs, n)
      else
        x :: loop(xs, i - 1)
    case Nil => Nil
}. ensuring{_ => _ == rec(i,n,ls)}
```

```
def rec(actual: Int, b: Int, ls: List[Int]) =
  require((actual >= 1) && (b > 0))
  ls match
    case Nil => Nil
    case first :: next =>
      if (actual == 1) then
        rec(b, b, next)
      else
        first :: rec(actual - 1, b, next)
```

- For recursive functions, proofs by induction

Background: Functional Induction in Stainless

```
def loop(ls: List[Int], i: Int) = {
  require(i > 0 && n > 0)
  ls match
    case x :: xs =>
      if i == 1 then
        val res = UNFOLD(loop(xs, n, n))
        assume(res == rec(n, n, xs))
        res
      else
        val res = UNFOLD(loop(xs, i - 1, n))
        assume(res == rec(i - 1, n, xs))
        xs :: res
    case Nil => Nil
}. ensuring{_ => _ == rec(i,n,ls)}
```

- Function unfolding and functional induction

Background: Functional Induction in Stainless

```
def loop(ls: List[Int], i: Int) = {
  require(i > 0 && n > 0)
  ls match
    case x :: xs =>
      if i == 1 then
        val res = UNFOLD(loop(xs, n, n))
        assume(res == rec(n, n, xs))
        res
      else
        val res = UNFOLD(loop(xs, i - 1, n))
        assume(res == rec(i - 1, n, xs))
        xs :: res
    case Nil => Nil
}. ensuring{_ => _ == rec(i,n,ls)}
```

- Function unfolding and functional induction

Our Experiment: Stainless as a Grading Service

- Second year course on Software Construction (Functional Programming)
- Four programming exercises, one reference solution per exercise
- 200 participants, 719 submissions

Our Experiment: Stainless as a Grading Service

- Second year course on Software Construction (Functional Programming)
- Four programming exercises, one reference solution per exercise
- 200 participants, 719 submissions

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

Feedback

Feedback summary directly available to the students

- Congratulations message – the submission was proven correct (equivalent to the reference solution)
- Counterexample errors – a counterexample input was found, proving that the submission is incorrect (not equivalent to the reference solution)

Feedback

Feedback summary directly available to the students

- Congratulations message – the submission was proven correct (equivalent to the reference solution)
- Counterexample errors – a counterexample input was found, proving that the submission is incorrect (not equivalent to the reference solution)
- User errors – incorrectly named file or an incorrect function signature
- Safety errors – such as division by zero or integer overflow

Feedback

Feedback summary directly available to the students

- Congratulations message – the submission was proven correct (equivalent to the reference solution)
- Counterexample errors – a counterexample input was found, proving that the submission is incorrect (not equivalent to the reference solution)
- User errors – incorrectly named file or an incorrect function signature
- Safety errors – such as division by zero or integer overflow
- Warnings – equivalence could not be proven within the timeout

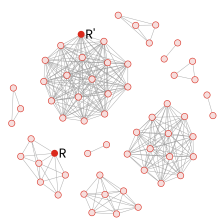
Feedback

Feedback summary directly available to the students

- Congratulations message – the submission was proven correct (equivalent to the reference solution)
- Counterexample errors – a counterexample input was found, proving that the submission is incorrect (not equivalent to the reference solution)
- User errors – incorrectly named file or an incorrect function signature
- Safety errors – such as division by zero or integer overflow
- Warnings – equivalence could not be proven within the timeout

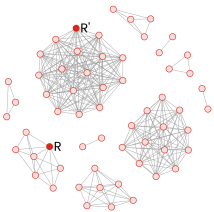
Additional feedback on the instructor side

- Clusters of submissions proven equivalent
- Singleton submissions
  (not provably equivalent to any other submission)

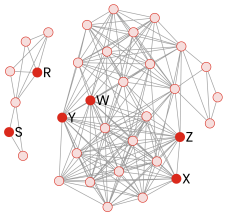


Results

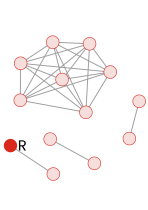
Name	S	TS	I	C	TO	Non-Singleton	Singleton
drop	60	56	169	14	70	10	26
gcd	21	2	8	3	42	2	12
prime	146	9	40	1	22	4	11
infix	2	0	2	42	0	1	0



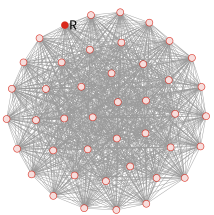
drop



gcd

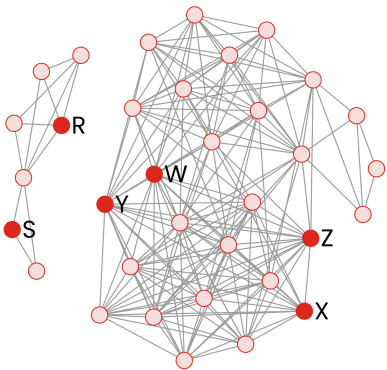


prime



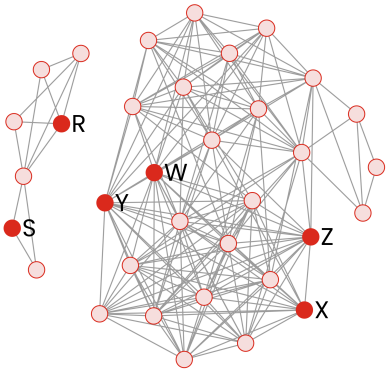
infix

Results: gcd



- 76 submissions
- 8 proven incorrect
- 3 proven correct
- 42 unknown result (timeout)
  - 2 non-singleton clusters
  - 12 singleton clusters

Results: gcd

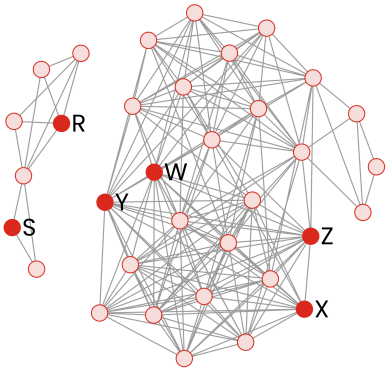


Submissions in the smaller cluster use the subtraction-based Euclid's algorithm.

```
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)
```

Results: gcd



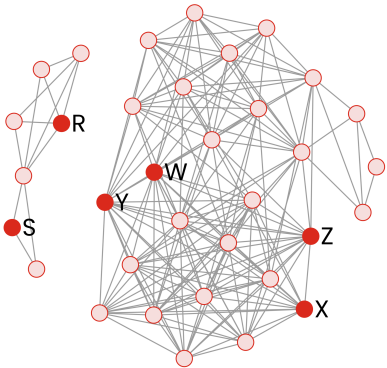
Submissions in the larger cluster use the modulo-based Euclid's algorithm.

```
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 =
  require(a >= 0 && b >= 0)
  b match
    case 0 => a
    case _  => gcdX(b, a % b)

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

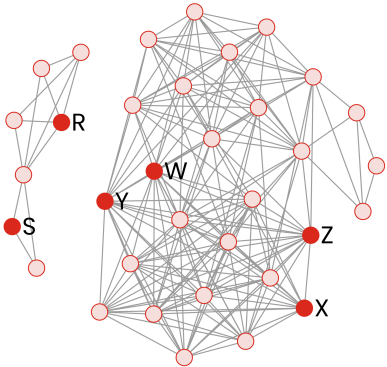
Results: gcd




```
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
```

Clustering does not differentiate purely functional programs from programs with loops and mutations.

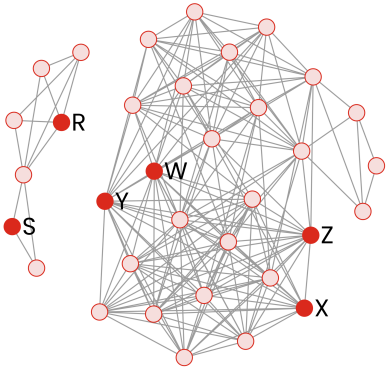
Results: gcd



Singleton clusters (not provably equivalent to any other submission) require further manual inspection.

```
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)
```

Results: gcd



Singleton clusters (not provably equivalent to any other submission) require further manual inspection.

```
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) // computes b
    gcd(q, r)
```

Takeaways

- A single reference solution is not sufficient to capture the **algorithmic variety** in student submissions.
- By failing to automatically classify certain supposedly correct submissions, the grader can help detect **suboptimal solutions**.
- Upon **manual inspection**, correct submissions can be promoted to reference solutions (one manual check per cluster).
- Formal verification provides **feedback** even for errors that cannot be formulated easily as counterexamples.
- Support for **library functions** is crucial. It allows exercising functional programming abstractions and leads to more diversity in submissions.

Resources

Accompanying artifact with all our benchmarks:
https://doi.org/10.5281/zenodo.14624668

Additional examples:
github.com/epfl-lara/stainless/tree/main/frontends/benchmarks/equivalence

Further Resources and Related Work

- Automated grading
  - Comparison to reference solution: AskElle (for Haskell), ZEUS (for Ocaml)
  - Clustering: OverCode, Clara, CoderAssist, LEGenT
  - LearnML: counterexample + feedback generation (FixML, TestML, CAFE)
- Equivalence checking
  - Regression verification: REVE, RVT
  - Translation validation
- Automated proofs by induction
  - Recursion induction in Isabelle, functional induction in Coq
  - Functional induction is the default induction heuristic in ACL2

Conclusions and Future Work

- Practical and rigorous autograding using program equivalence checking
- Formal verification complements testing-based grading and enriches the feedback given to students
- Differentiates solutions, even those with identical input-output behaviour
- In the future: progressively use the approach on more (larger) exercises
- So far: well-defined input-output; Future work: underspecified assignments