Proving and Disproving Equivalence of Functional Programming Assignments

DRAGANA MILOVANČEVIĆ, EPFL, Switzerland
VIKTOR KUNČAK, EPFL, Switzerland

We present an automated approach to verify the correctness of programming assignments, such as the ones that arise in a functional programming course. Our approach takes as input student submissions and reference solutions, and uses equivalence checking to automatically prove or disprove correctness of each submission. To be effective in the context of a real-world programming course, an automated grading system must be both robust, to support programs written in a variety of style, and scalable, to treat hundreds of submissions at once. We achieve robustness by handling recursion using functional induction and by handling auxiliary functions using function call matching. We achieve scalability using a clustering algorithm that leverages the transitivity of equivalence to discover intermediate reference solutions among student submissions. We implement our approach on top of the Stainless verification system, to support equivalence checking of Scala programs. We evaluate our system and its components on over 4000 programs drawn from a functional programming course and from the program equivalence checking literature; this is the largest such evaluation to date. We show that our system is capable of proving program correctness by generating inductive equivalence proofs, and providing counterexamples for incorrect programs, with a high success rate.

CCS Concepts: • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: equivalence checking, functional induction, automated grading

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

1 INTRODUCTION

Program verification has made great advances in recent years, resulting in tools that can formally verify strong correctness properties of important pieces of software infrastructure [Chajed et al. 2017; Heiser et al. 2020; Klein et al. 2018]. At the core of our paper is a particular instance of program verification: equivalence checking. An appealing aspect of this problem is that the specification itself is an executable program that is often naturally available in a given application scenario. In addition to applications in regression verification [Felsing et al. 2014; Strichman and Godlin 2005] and translation validation [Gupta et al. 2020], program equivalence naturally arises when grading programming assignments [Song et al. 2019]. The equivalence problem is undecidable in general, even if interesting restricted cases of the problem are decidable by reduction to string transducers [Alur and Černý 2011].

To see why proving equivalence is important in the context of grading, we observe that even very simple examples can lead to misleading messages to students from automated grading tools, or

Authors' addresses: Dragana Milovančević, EPFL, Switzerland, dragana.milovancevic@epfl.ch; Viktor Kunčak, EPFL, Switzerland, viktor.kuncak@epfl.ch.

2475-1421/2023/6-ART144
https://doi.org/10.1145/3591258

```
sealed abstract class Nat

case object ZERO extends Nat
case class SUCC(param0: Nat) extends Nat

def n2i(x: Nat): Int = x match
  case ZERO => 0
  case SUCC(y) => 1 + n2i(y)

def i2n(x: Int): Nat = x match
  case 0 => ZERO
  case _ => SUCC(i2n(x − 1))

def natadd(n1: Nat, n2: Nat) : Nat =
  i2n(n2i(n1) + n2i(n2))
```

(a) Buggy addition of user-defined natural numbers. The original OCaml implementation appears in several submissions in an actual functional programming course.

```
type Binary = List[Boolean]

def b2i(l: Binary): Int = l match
  case Nil() => 0
  case Cons(true, tl) => 1 + 2 ∗ b2i(tl)
  case Cons(false, tl) => 2 ∗ b2i(tl)

def i2b(x: Int): Binary = x match
  case 0 => List(false)
  case _ =>
    if (x % 2 == 1) true :: i2b(x/2)
    else false :: i2b(x/2)

def binadd(n1: Binary, n2: Binary) =
  remove0s(i2b(b2i(n1) + b2i(n2)))
```

(b) Buggy addition of binary representations of natural numbers. The bits in the linked list are stored starting from least significant digit to the most significant digit.

Fig. 1. Subtle bugs in introductory programming exercises.

even from instructors. While this is a well-known problem in test suite design [Wrenn et al. 2018], it is also present in tools that perform automated error detection. For example, Figure 1a shows a student submission for an assignment on addition of user-defined natural numbers. This program converts numbers from the unary representation into a bounded, signed integer type. The original OCaml program comes from a publicly available benchmark from the LearnML framework [Song et al. 2021], where it appears in several students' submissions. TestML [Song et al. 2019], a state of art tool for automatic detection of logical errors in programming assignments, does not find a counterexample when comparing this submission to a correct reference solution. The submission also passes all the test cases provided by the authors and is classified as correct. However, when given a number in unary of length 2^{31}, this program would first convert it into a machine integer and overflow in the process, so the result of adding two numbers would be a smaller unary number. In some cases, we could have negative numbers on which the conversion would result in an infinite loop. Finding such counterexamples involves data structure input of length 2^{31} and symbolic execution paths of this length, so it is not surprising that testing or symbolic execution do not identify such cases. While this example might seem benign given that it requires unrealistically large inputs to demonstrate these bugs, it is not an isolated case. Consider a more memory-efficient representation of natural numbers: a binary representation stored as a list of bits. Figure 1b shows an incorrect submission that performs addition of binary representations by converting them to built-in integers and back. Running TestML on this example against a reference solution, once again, results in the absence of counterexamples.

Classifying non-equivalent programs as equivalent (and consequently correct) merely based on the absence of counterexamples is problematic, as it fails to educate students in rigorous reasoning about software. We thus propose the student assignment equivalence challenge: *Develop an effective and practical technique that takes realistic sets of submissions to introductory functional programming assignments and proves each equivalent to a correct solution.*

Checking student submissions is a broader problem than equivalence checking of two functions, for several reasons. First, there is a large number of submissions. An equivalence checking system needs to check all candidate submissions against reference solutions. This aspect emphasizes the importance of equivalence checking and makes the problem an interesting combination of program verification and clustering [Gulwani et al. 2018; Singh et al. 2013]. Second, it is a situation where counterexample generation is helpful, to convince students of their mistakes (as symbolic execution techniques do [Song et al. 2019]) and proofs are likewise desirable, both for fairness of grading, and to avoid giving false confidence to students (and instructors), as illustrated in our examples from Figure 1. Third, even in introductory programming classes, students learn to write modular programs, composed of multiple recursive functions, such as the programs from our motivating examples (Figure 4). Proving the equivalence of such programs uncovers the problem of matching equivalent auxiliary functions [Godlin and Strichman 2013]. Finally, instructors have the ability to provide multiple reference solutions that have, for example, different recursion schemas, resulting in a problem of proving equivalence of each of the (many) candidate submissions against one of the (few) reference solutions.

In this paper, we propose a new approach for verifying the correctness of functional programming assignments, using automated equivalence checking, as well as a clustering algorithm to efficiently treat many candidate submissions at once. Our approach relies on automated functional induction, which is, in principle, applicable to a variety of scenarios. Our system emits warnings for possible addition overflows in the program from Figure 1a and reports a list of size 35 as a counterexample for program from Figure 1b. Even more importantly, since our approach only classifies programs as correct based on the outcome of verification, it provides a guarantee that programs are never wrongly classified as correct, or incorrect, reducing the danger of misleading students.

The main contributions of this paper are:

- **Automated function equivalence proofs:** We propose the use of induction on the recursive call structure as the strategy of choice for checking the equivalence of a student submission against a reference solution.
- **Function call matching:** We present a new technique for function call matching for equivalence checking of programs that contain auxiliary recursive functions. Our technique uses type- and test-directed search to find pairs of equivalent auxiliary functions.
- **Clustering based on equivalence:** We introduce a new algorithm for clustering programs based on equivalence checking and counterexample evaluation, with dynamic prioritization of reference solutions. We leverage the transitivity of equivalence by using programs that are already proven correct to automatically discover intermediate reference solutions.
- **Implementation and evaluation:** We implement our approach on top of Stainless, an existing open-source verifier for Scala [EPFL 2023]. We evaluate our system on over 4000 submissions from 15 functional programming assignments, as well as 17 benchmarks from the equivalence checking literature. We contribute a public Scala version of our data set. We show that our system is effective in both automatically constructing proofs and finding meaningful counterexamples. We are not aware of any system for automated formal equivalence checking of recursive programs that achieves comparable results.

2 OVERVIEW

In this section, we give an overview of our approach on two example assignments from functional programming courses. The first example is an introductory assignment, to illustrate the underlying inductive proof generation mechanism. The second example is an assignment with multiple auxiliary recursive functions, to illustrate our function call matching techniques.

```
def isSortedR(l: List[Int]): Boolean =
  def loop(p: Int, l: List[Int]): Boolean = l match
    case Nil() => true
    case Cons(x, xs) if (p <= x) => loop(x, xs)
    case _ => false
  if (l.isEmpty) true
  else loop(l.head, l.tail)
```

```
def isSortedB(l: List[Int]): Boolean =
  if (l.isEmpty)
    true
  else if (!l.tail.isEmpty && l.head > l.tail.head)
    false
  else
    isSortedB(l.tail)
```

```
def isSortedA(l: List[Int]): Boolean =
  def leq(cur: Int, next: Int): Boolean = cur < next
  def iter(l: List[Int]): Boolean =
    if (l.isEmpty) true
    else if (l.tail.isEmpty) true
    else leq(l.head, l.tail.head) && iter(l.tail)
  if (l.size < 2) true
  else l.head <= l.tail.head && iter(l.tail)
```

```
def isSortedC(l: List[Int]): Boolean =
  def chk(l: List[Int], p: Int, a: Boolean): Boolean =
    if (l.isEmpty) a
    else if (l.head < p) false
    else chk(l.tail, l.head, a)
  if (l.isEmpty) true
  else chk(l, l.head, true)
```

Fig. 2. Motivating example: reference solution *isSortedR* and three candidate submissions. Our technique proves the correctness of *isSortedB* and *isSortedC*, by proving them equivalent to the reference solution. For *isSortedA*, our technique disproves the equivalence and reports a counterexample. The only inputs that our tool requires are the source files and the names of candidate and reference functions.

Example 1: Check Whether a List is Sorted. Consider the following programming assignment: *Write a function isSorted that checks whether a given list of integers is sorted in a non-decreasing order.* This assignment is part of a graded lab of an introductory Scala course, which introduces students to Scala's syntax and the basics of functional programming, such as immutability and recursion. Figure 2 shows the reference solution (isSortedR) and three candidate submissions (isSortedA, isSortedB and isSortedC), motivated by students' submissions. Candidate submissions are structurally different from the reference solution, and among themselves: they use zero (isSortedB), one (isSortedR, isSortedC), and two (isSortedA) inner functions, with different numbers of arguments. Candidate submissions are also syntactically different from the reference solution: they use a combination of if-then-else and list functions, instead of pattern matching. Control flow also differs slightly among submissions: branches appear in a different order and have different conditions.

To prove (or disprove) the correctness of submissions, we use reference solutions as a specification, which defines the correct behavior. In other words, an instructor only provides a correct solution, no additional specification or annotations are required. To prove the equivalence of recursive functions, we use proofs by induction on the execution trace of one of the functions. Figure 3 illustrates the automatic transformations that take place in our system to encode proofs by induction, for functions isSortedB and isSortedR from Figure 2. We use the recursive function isSortedB as a template for the inductive proof, where the *ensuring* statement specifies a contract that the function must satisfy. We expand recursive calls by defining the function to be equal to one unfolding of its body, keeping the *ensuring* clause as an assumption. After one unfolding, we derive formulas corresponding to the program from Figure 3b. If the execution takes the recursive branch, and assumes that the postcondition holds for xs, then it also holds for the input list l. This combination of *assume*/*ensuring* statements, which we obtain as a result of function unfolding, corresponds to an inductive step in our proof.

```
def isSortedB(l: List[Int]): Boolean = {
  if (l.isEmpty) true
  else if (!l.tail.isEmpty && l.head > l.tail.head)
    false
  else isSortedB(l.tail)
} ensuring(_ == isSortedR(l))
```

(a) The *ensuring* statement specifies a contract that the function must satisfy.

```
def isSortedB(l: List[Int]): Boolean = {
  if (l.isEmpty) true
  else if (!l.tail.isEmpty && l.head > l.tail.head)
    false
  else
    val xs = l.tail
    val res =
      if (xs.isEmpty) true
      else if (!xs.tail.isEmpty && xs.head > xs.tail.head)
        false
      else isSortedB(xs.tail)
    assume(res == isSortedR(xs))
    res
} ensuring(_ == isSortedR(l))
```

(b) After one unfolding, the recursive call maps to an inductive step in our proof.

Fig. 3. Unfolding and functional induction: checking the equivalence of *isSortedB* and *isSortedR* (Figure 2)

Our system successfully proves the equivalence of isSortedB and isSortedR using induction. For isSortedC, the equivalence checking against isSortedR times out. Subsequently, however, induction succeeds at proving the equivalence of isSortedC and isSortedB. By transitivity, our approach concludes that isSortedC is also equivalent to isSortedR, and therefore correct. On the other hand, our system labels isSortedA as incorrect, and reports a counterexample: Cons(0, Cons(0, Cons(0, Nil()))).

Our technique automatically uses submissions that are already proven correct as intermediate reference solutions. In our evaluation (Section 5.6), we show how processing multiple submissions at the same time improves the success of equivalence checking in practice, thanks to automatic discovery of intermediate reference solutions.

Example 2: Get Unique Values From a List. We now illustrate our pairwise equivalence checking techniques in the presence of auxiliary recursive functions, on an example assignment from LearnML's benchmarks [Song et al. 2021]. Figure 4 shows two programs that implement removal of redundant elements in a list: our reference solution uniqR and a student submission uniqA. Our pairwise equivalence subroutine first generates an equivalence lemma to directly check the equivalence of uniqR and uniqA. This verification attempt times out due to the absence of an induction schema in uniqR and uniqA's bodies. We thus proceed to try matching equivalent function calls: we first identify function calls in uniqR and uniqA, and then select the best candidate match for each call in uniqR. This shapes our verification attempt into proving the following:

- the equivalence of find and isin,
- the equivalence of unique and distinct, with calls to isin replaced by calls to find and
- the equivalence of uniqR and uniqA, with calls to distinct replaced by calls to unique.

The last proof checks the equivalence of two non-recursive functions, and therefore the problem is simply mapped into SMT queries. For the first and second proof, we use functional induction on find and unique, respectively. This results in a successful equivalence proof.

Our technique automatically matches function calls in order to prove program equivalence. To discover candidate matching pairs of functions, we start by identifying signature-compatible functions. In case of multiple signature-compatible candidates, we use testing to further narrow down

```
def find(lst: List[Int], n: Int): Boolean = lst match
  case Nil() => false
  case Cons(hd, tl) => (n == hd) || find(tl, n)

def unique(l: List[Int], r: List[Int]): List[Int] =
  l match
    case Nil() => r
    case Cons(hd, tl) =>
      if (!find(r, hd)) unique(tl, r ++ List(hd))
      else unique(tl, r)

def uniqR(lst: List[Int]): List[Int] =
  unique(lst, Nil())
```

```
def isin(a: Int, lst: List[Int]): Boolean =
  lst.foldRight(false){ (e, acc) =>
    (e == a || acc) }

def distinct(a: List[Int], b: List[Int]): List[Int] =
  b match
    case Nil() => a
    case Cons(hd, tl) =>
      if (isin(hd, a)) distinct(a, tl)
      else distinct(a ++ List[Int](hd), tl)

def uniqA(lst: List[Int]): List[Int] =
  distinct(List(), lst)
```

Fig. 4. Example programs to illustrate function call matching. Our pairwise equivalence checking subroutine proves *uniqA* equivalent to *uniqR*, by matching functions *unique* and *distinct*, as well as $find$ and *isin*.

the function matching search space. Our test cases are the counterexamples we obtain from treating many candidate submissions at once, which keeps our approach fully automatic.

In our example from Figure 4, matching find and isin is straightforward based on type signatures; in particular only the function find has the type signature compatible with the signature of isin. To match unique and distinct, there are two possible alternatives with respect to signature compatibility: one possibility is to keep the original argument order, and the other is to swap the argument order. By testing both options, we quickly find that the original order is not a good match, and the swapped order is a better candidate for our equivalence conjecture. Our system always follows up on those conjectures with a formal verification step: we only use tests to narrow down candidate matching pairs of functions, never as a stand-alone criterion for correctness.

3 EQUIVALENCE CLUSTERING ALGORITHM

In this section, we discuss the algorithm we designed to check the correctness of a set of student submissions against a set of reference solutions. We present the following main ideas:

- Pairwise equivalence checking based on functional induction
- Function call matching based on type- and test-directed search
- Our clustering algorithm that leverages transitivity and counterexamples to determine the subset of correct submissions, using pairwise equivalence as a subroutine.

3.1 Language and Terminology

A program is defined by its entry point function, and might contain auxiliary (inner or separate) functions. We consider purely functional terminating programs, without side effects. Our language supports recursive functions, higher-order functions, type parameters, basic built-in types (Int, BigInt, Char, Boolean), List, Pair and user-defined types. In our language, an *ensuring* statement specifies a function's postcondition. An *assume* statement implies that the associated property holds. Our language does not have exceptions.

A reference solution (teacher solution) is a correct, terminating program. A candidate submission (student submission) is a program that we are trying to prove correct. A candidate submission f is equivalent to a reference solution m if:

- m and f have the same signature,
- m and f terminate and
- m and f return the same output for all inputs.

The termination requirement is a consequence of our choice of induction as the main proof technique (Section 3.2). In the literature on automated induction and equivalence checking, the termination requirement is often put aside and left as a responsibility of users [Badihi et al. 2020; Claessen et al. 2012; Felsing et al. 2014; Wood et al. 2017]. This practice opens the door to a student submission subverting the equivalence checker into incorrectly claiming the equivalence of non-equivalent non-terminating programs. While automated termination checking is out of scope of our work, we evaluate and discuss the implications of the termination requirement in Section 5.2.

3.2 Pairwise Equivalence Checking Using Automated Functional Induction

To prove the equivalence of recursive functions, we use proofs by induction derived from the recursion structure of one of the functions. The idea that a terminating function defines a well-founded relation usable as an inductive principle goes at least as far back as [McCarthy 1963]. Such induction principles are available in many interactive proof systems. In ACL2 [Kaufmann et al. 2000] they are the default induction heuristic, in Isabelle/HOL they are called *recursion induction* [Nipkow et al. 2002] or *computation induction* [Nipkow 2022] (denoted by f.induct for a function f), whereas in Coq they are called *functional induction* [INRIA 2021].

Consider the following programs f1 and f2, with identical parameter lists params and identical return type ParamTypes, whose bodies are expressions F1 and F2, respectively:

```
def f1(params: ParamTypes): RetType = F1
def f2(params: ParamTypes): RetType = F2
```

The following lemma states the equivalence of f1 and f2:

```
def lemma(params: ParamTypes): Unit = {} ensuring(f1(params) == f2(params))
```

In our approach, when functions are not recursive, we directly encode equivalence lemmas into SMT solver formulas. When one of the functions is recursive, we generate an inductive proof based on that function's definition. Intuitively, such induction uses recursive calls as induction steps and branches without recursive calls as base cases.

Consider the case when function f1 is recursive and suppose its body F1 is of the form G1[params, f1(H1[params])], where G1 and H1 denote expressions with holes. We define indProof to be a copy of f1 with the lemma's postcondition attached to it, and state the main lemma as follows:

```
def lemma(params: ParamTypes): Unit = {
  val proof = indProof(params)
} ensuring(f1(params) == f2(params))
```

```
def indProof(params: ParamTypes): RetType = {
  G1[params, indProof(H1[params])]
} ensuring(f1(params) == f2(params))
```

The main lemma succeeds trivially. For indProof, this encoding is effective thanks to body-visible recursion [Hamza et al. 2019, Section 5.3] and function unfolding [Suter et al. 2011]. Body-visible recursion enables the verifier to examine a function definition while proving its properties. Function calls are initially treated as uninterpreted functions; each call is then iteratively expanded to an unfolding of the function's body, with an assume statement matching the original postcondition.

After one unfolding, the recursive call indProof(H1[params]) becomes:

```
G1[H1[params], indProof(H1[H1[params]])]
assume(f1(H1[params]) == f2(H1[params]))
```

With this assumption that the equivalence proof holds for the recursive call indProof(H1[params]), we can prove that it holds for indProof(params).

3.3 Function Call Matching Based on Type- and Test-Directed search

We now extend the pairwise equivalence checking problem to detect calls to distinct but equivalent (auxiliary) functions. This problem is a variant of RVT's decomposition algorithm [Godlin and Strichman 2013], where we additionally consider argument permutations for auxiliary functions. When proving the equivalence of programs that contain calls to equivalent functions we can replace calls to one function with calls to the other. The goal of function call matching is to identify and substitute such calls.

Equivalence Modulo Equivalent Auxiliary Functions. We first show how to generalize the equivalence problem if two functions make calls to equivalent auxiliary functions. Consider the case when f1 and f2 contain calls to functions c1 and c2, such that:

- f1 calls c1, where c1 ≠ f1,
- f2 calls c2, where c2 ≠ f2,
- c1 and c2 have compatible signatures, that is, identical return types and identical multisets of argument types.

In other words, we can think of the bodies of f1 and f2 as:

```
F1 = U1[params, c1(V1[params])]
F2 = U2[params, c2(V2[params])]
```

where U1 and U2 denote expressions with holes and V1[params] and V2[params] lists of arguments.

When c1 and c2 are equivalent for some argument permutation σ that aligns the two signatures, we can substitute calls to c2 with calls to c1 in the body of f2, obtaining a new body:

```
F2' = U2[params, c1($\sigma$[V2[params]])]
```

To prove the equivalence of f1 and f2, it then suffices to prove the equivalence of f1 and f2', where f2' is defined as:

```
def f2'(params: ParamTypes): RetType = F2'
```

As a result of function call matching, we obtain the following lemmas that state the equivalence of f1 and f2, while taking into account structural similarities between the two programs:

```
def sublemma(params': ParamTypes'): Unit = {
  val proof = indSubProof(params')
} ensuring(c1(params') == c2($\sigma^{-1}$[params']))
```

```
def lemma(params: ParamTypes): Unit = {
  val proof = indProof(params)
} ensuring(f1(params) == f2'(params))
```

Here, indSubProof and indProof are built as in Section 3.2, from the bodies of c1 and f1, respectively. We can do this matching for arbitrary many function calls in f1 and f2, and recursively for equivalent matched functions. We next outline how to find pairs of functions c1, c2.

Search for Equivalent Auxiliary Functions. Our search for pairs of equivalent auxiliary functions c1 and c2 and the corresponding argument permutation σ is based on the following criteria:

- **type-directed search**: We account for every pair of functions with compatible signatures and every argument permutation that results in parameters with identical types.
- **test-directed search**: We perform test-directed search to narrow down the previous selection. We use the counterexamples found by our system to filter out function pairs that evaluate to different results, and are thus provably non-equivalent.

For functions with distinct argument types, type-directed search alone does a good job at finding candidate pairs. When type-directed search yields more than one alternative, test-directed search helps pruning the search space.

3.4 Clustering Algorithm

The goal of our clustering algorithm is to classify candidate submissions by comparing them against reference solutions, using pairwise equivalence as a subroutine. The algorithm takes as input a set of candidate submissions and a set of reference solutions. The output of the algorithm is a classification of candidate submissions into the following categories:

- **wrong**, if the signature is wrong;
- **correct**, if there is a reference solution provably equivalent to this program;
- **incorrect**, if there is a counterexample that disproves the equivalence;
- **unknown**, when there is neither a proof of equivalence nor a counterexample.

All candidate submissions are initially labeled as "unknown". The algorithm classifies each candidate submission f through the following steps:

(1) Inspect the candidate signature to make sure that it corresponds to the given reference signature. If not, label f as "wrong" and do not attempt to construct an equivalence proof.
(2) Assuming that f has the correct signature, evaluate it on the current tests. In case of a failure, label f as "incorrect".
(3) Otherwise, proceed with verification: for each reference solution m from the top N reference solutions, attempt to prove the equivalence between f and m.
    (a) If the two programs are equivalent, label f as "correct" and store m as its reference solution. Then add f itself to the priority queue of reference solutions.
    (b) If the two programs are not equivalent, label f as "incorrect" and store the counterexample for which the two functions give different outputs.
    (c) If the verification times out, go back to step 3 and pick another reference solution. If the top N reference solutions are exhausted, the iteration concludes and f remains labeled as "unknown".
(4) If there are no further candidate submissions to process, repeat the previous steps for the submissions that remain "unknown", until reaching a fixpoint.

The algorithm uses a priority queue to store reference solutions. Each time a candidate submission is proven correct, is it added to this queue. Priorities are based on a simple heuristic: programs are rewarded every time they are used to prove correctness, and penalized every time an equivalence proof times out. The algorithm compares every candidate submission against the top N reference solutions according to this heuristic, where N is a parameter of the algorithm. Section 5.6 analyzes the impact of our heuristic and the N parameter.

The algorithm uses two subroutines: one for testing, and one for verification. Test cases in our system consist of two groups: counterexamples obtained from disproving equivalence of incorrect candidate submissions, and optional (not used in our evaluation) user-defined tests. The verification subroutine generates and verifies equivalence lemmas as described in Sections 3.2 and 3.3. It generates up to four verification attempts: with and without trying to match equivalent function calls, once using the reference solution, and once using the candidate submission as a template for induction.

4 IMPLEMENTATION

We implement our approach on top of the Stainless verification system [Hamza et al. 2019; Voirol 2019]. Stainless is a tool for verifying programs written in a subset of the Scala programming language [Odersky et al. 2019], relying on an SMT solver. It can be configured to use Z3 [De Moura and Bjørner 2008], which we used in our experiments, CVC4 [Barrett et al. 2011], or Princess [Rümmer 2008]. Stainless takes as input Scala programs, with optional specification, and can verify their correctness, but also report counterexamples when they exist.

Internally, Stainless operates on abstract syntax trees, that are transformed throughout several pipeline phases. This series of transformations results in verification conditions that are then sent to an SMT solver. To implement equivalence proof generation, we add a new phase to the Stainless verification pipeline. Our phase is responsible for generating equivalence lemmas and all the associated transformations. The entire process, including defining and proving equivalence lemmas, is fully automatic, following the techniques presented in Section 3.

Our approach is not specific to Stainless as such, but relies on the ability to explicitly encode inductive proofs through postconditions of terminating recursive functions. In addition, the efficiency of our algorithm benefits from the ability of the underlying tool to return counterexamples for non-equivalent programs. Performance-wise, our system benefits from Stainless using a cache for verification queries, which reduces the number of solver calls [Guilloud et al. 2023].

5 EVALUATION

In the first part of our evaluation, we assess the practical value of our system in the context of grading. To this end, we collect over 4000 submissions from 15 functional programming assignments, from publicly available automated grading benchmarks. In this section, we first introduce our data set (Section 5.1), and explain our preliminary termination analysis (Section 5.2). We then present the main results of our evaluation (Section 5.3) and compare our results to TestML (Section 5.4). We show that our system is effective in both automatically constructing proofs and finding meaningful counterexamples, with a 86% success rate. We additionally demonstrate how each aspect of our equivalence checking contributes to the overall results (Section 5.5) and discuss the performance of our clustering algorithm (Section 5.6).

In the second part of our evaluation, we consider 17 benchmarks from the equivalence checking literature (Section 5.7). We show that our system overcomes the limitations of existing tools, with a 82% success rate on these benchmarks.

We conclude by discussing practical implications of our results in the context of automated grading (Section 5.8).

5.1 Benchmark Setup

We evaluate our system on publicly available benchmarks from the LearnML framework [Lee et al. 2018; Song et al. 2019, 2021]. These benchmarks consist of OCaml assignments, collected over several iterations of a programming languages course. All the submissions are syntactically correct, and without type errors, but might contain logical errors. Table 1 describes these assignments, along with the number of our reference solutions and total number of submissions per assignment. The assignments are typical introductory programming problems, and include user-defined types and higher-order functions, including library higher-order functions such as foldLeft.

To evaluate our system on the OCaml benchmarks, we used a semi-automated translator from OCaml to Scala. Our translator first adds type annotations to OCaml programs using the ocamlc compiler [INRIA 2007a]. It then parses the resulting programs using Camlp5 [INRIA 2007b] and translates them to Scala source files using a custom pretty printer, implemented within the Camlp5 framework. Our translator supports type declarations, exceptions, variable definitions and expressions. Since our approach does not consider exceptions, we translated exceptional behavior using sentinel values. For cases where our simple automated translation did not result in valid Scala programs, we manually fixed those translations. Examples of manual interventions include: calls to library functions, partial function applications, type annotations for inner functions and missing cases for pattern matching exhaustiveness. We exceptionally adapted the lambda benchmark to use Integers instead of Strings, without affecting the program semantics. This change allows us to systematically use Z3 for all the benchmarks (the version of these programs with Strings requires

Table 1. Description of our benchmark programs. #S and #P indicate the number of reference solutions and candidate submissions for each problem, respectively. #F shows the average number of functions in candidate submissions. The last column shows the average size of programs in number of lines of code.

No	Name	Description	#S	#P	#F	LOC
1	filter	Filtering elements satisfying a predicate in a list	1	210	1.1	9
2	max	Finding a maximum element in a list	1	216	1.5	15
3	mirror	Mirroring a binary tree	1	97	1.1	19
4	mem	Checking membership in a binary tree	1	136	1.1	18
5	sigma	Computing $\sum_j^k f(i)$ for j, k, and f	3	736	1.1	11
6	natadd	Adding user-defined natural numbers	4	447	1.4	13
7	natmul	Multiplying user-defined natural numbers	7	447	2.9	27
8	change	Finding the number of ways of coin-changes	1	9	2.6	26
9	heap	Implementing a priority queue using a leftist heap	1	20	6.4	39
10	uniq	Removing redundant elements in a list	4	157	2.8	23
11	iter	Composing functions	1	28	1.3	11
12	crazy2add	Adding numbers in user-defined number system	1	240	2.0	52
13	formula	Evaluating expressions and propositional formulas	1	708	2.1	52
14	lambda	Deciding if lambda terms are well-formed or not	4	802	2.6	34
15	diff	Differentiating algebraic expressions	1	409	2.1	60

using CVC4, with a larger timeout). The source code of all the translated programs, enhanced with termination annotations, is publicly available [Milovancevic and Kuncak 2023].

5.2 Termination Analysis

Termination checking is a prerequisite for equivalence proofs in our system. Stainless performs termination checking and requires all functions to have a measure, either inferred automatically or defined explicitly. In order to critically assess the practical value of our implementation on top of Stainless, this section presents the results of our termination analysis.

Table 2 shows the number of programs that we prove terminating, per benchmark.

Our termination analysis detects programs that are either provably non-terminating, or where measure inference fails. To better understand the cases where measure inference fails, consider the following (simplified) example from the formula assignment:

```
def eval(f: Formula): Boolean = f match
  case True => true
  case False => false
  case Not(f) => if (eval(f)) false else true
  case Imply(fl, fr) => eval(Not(fl)) || eval(fr)
```

This assignment uses a user-defined type Formula. For the recursive call eval(Not(fl)), Stainless fails to automatically infer that the size of the argument decreases. In our translation, we thus had to define a size for the type Formula that attributes a higher cost to the Imply node, and specify it as a measure explicitly, using the decreases annotation: **decreases**(f.size).

The limitation of Stainless' current measure inference requires even more manual interventions in the case of crazy2add and diff benchmarks, where automatic measure inference fails even for the reference solutions, and only a few programs were automatically proven terminating (5% for the crazy2add benchmark). We therefore exclude crazy2add and diff benchmarks from further analysis.

We also added explicit measure annotations for the sigma benchmark, including the reference solution. As the resolving annotation is rather systematic, we could easily apply it directly in our translation. For the rest of benchmarks, we either specified trivial decreases annotations for a small percentage of submissions (change: 33%, uniq: 30%, lambda: 10%, formula: 4%), or fully relied on Stainless to infer measures automatically (filter, max, mirror, mem, natadd, natmul, heap, iter).

5.3 Equivalence Checking Results

Table 2 shows the results of our evaluation. Our system manages to prove or disprove equivalence for 86% of the submissions that we prove terminating (3292 out of 3808). We attribute the high success rate to our use of functional induction and to Stainless systematically checking not only for proofs but also counterexamples in each unfolding [Voirol et al. 2015].

We were not able to evaluate our system on the iter benchmark due to limitations of Stainless, which uses referential equality to compare function literals. The iter assignment is about composing functions and the solution requires returning function literals, so given such notion of equality, our system cannot prove equivalence of those programs. We exclude crazy2add and diff benchmarks from the equivalence checking analysis because we could not prove termination for the majority of those submissions, as explained in Section 5.2.

5.4 Comparison to TestML

We compare our results to TestML [Song et al. 2019], a part of the LearnML framework for detecting logical errors in programming assignments. Our analysis relies on the evaluation report for TestML, for the subset of benchmarks that is presented in their report. On this subset, TestML finds more errors than hand-written test cases, and there is not a single error which is detected by hand-written test cases but missed by TestML. Apart from being superior to manually-designed tests, their extensive evaluation shows that it also outperforms property-based testing and pure enumerative or symbolic approaches. For *disproving* the correctness of terminating programs, our approach is on par with TestML: our system reports counterexamples for 414 out of 420 incorrect submissions. Additionally, our system can *prove* the correctness of 2878 out of 3361 correct submissions, while TestML can provide no such formal correctness guarantees.

We additionally run TestML on the assignments that were omitted from their report: change and heap benchmarks, that were used in the prior tool, FixML [Lee et al. 2018], as well as the uniq and natadd/natmul benchmarks, that were used in the subsequent tool, CAFE [Song et al. 2021]. On the change and heap benchmarks, TestML fails to find counterexamples for 6/29 programs, which are all detected by the user defined test cases. For the change benchmark, interestingly, the provided reference solution was non-terminating. We had to modify the original OCaml solution to throw an exception for invalid inputs that were causing non-termination. While TestML only finds counterexamples for 6/9 incorrect programs, our system reports counterexamples for 8/8 terminating programs. For the heap benchmark, TestML finds counterexamples for 17/20 incorrect programs. Our system finds counterexamples for all 11 terminating programs. It detects counterexamples that disprove termination in 6 more submissions, and fails to infer measures for the remaining three programs (the same three programs for which TestML does not find counterexamples). We suspect that TestML does not find those counterexamples because of non-termination. The authors of TestML do not provide an explanation, as these benchmarks are excluded from their evaluation.

For the uniq benchmark, both TestML and our tool find counterexamples for all 31 terminating programs. For the natadd/natmul benchmark, TestML finds counterexamples for all the programs that are identified as incorrect by hand-written test cases. However, in the absence of counterexamples, it wrongly classifies certain programs with overflows as correct, as explained in the example from Section 1 (Figure 1a). Our tool either times out with warnings or detects overflows in those

Table 2. Evaluation results when running our benchmarks using Z3 with a 1 second timeout. The PT column shows the total number of programs that we prove terminating, per benchmark. The rest of the table shows the equivalence checking results for those programs (Column Found), as well as the results of TestML (Column TML). TO indicates the number of programs where our system times out. F/T indicates the percentage of programs where our system succeeds at proving or disproving the equivalence. The last three columns (NI, NM, 1RS) show the same percentage when disabling functional induction, function call matching, and multiple reference solutions, respectively.

Name	PT	Non-Equiv.			Equivalent			TO	F/T	NI	NM	1RS
		Found	TML	Total	Found	TML	Total					
filter	210	9	9	9	199	N/A	201	2	99%	4%	99%	99%
max	216	45	45	45	154	N/A	171	17	92%	21%	92%	92%
mirror	96	8	8	8	88	N/A	88	0	100%	8%	100%	100%
mem	136	19	19	19	117	N/A	117	0	100%	14%	100%	100%
sigma	734	30	30	30	682	N/A	704	22	97%	4%	95%	82%
natadd	381	2	2	2	351	N/A	379	28	93%	1%	93%	41%
natmul	381	29	29	29	209	N/A	352	143	62%	8%	8%	21%
change	8	8	6	8	0	N/A	0	0	100%	100%	100%	100%
heap	11	11	11	11	0	N/A	0	0	100%	100%	100%	100%
uniq	147	31	31	31	88	N/A	116	28	81%	21%	21%	50%
iter	27	–	–	–	–	–	–	–	0%	0%	0%	0%
crazy2add	–	–	–	–	–	–	–	–	–	–	–	–
formula	680	98	98	98	563	N/A	582	19	97%	14%	14%	97%
lambda	781	124	130	130	427	N/A	651	230	71%	16%	16%	40%
diff	–	–	–	–	–	–	–	–	–	–	–	–

cases. Therefore, even if it does not manage to disprove equivalence, it still separates these programs from the correct ones.

5.5 Ablation Study

We now evaluate the key elements of our equivalence checking, to better demonstrate how each element contributes to our results. To this end, we conduct an ablation study to further examine the impact of functional induction, function call matching, and multiple reference solutions.

Functional Induction. We use functional induction to prove the equivalence of recursive functions, as explained in Section 3.2. Given that all our benchmarks contain recursive functions, functional induction is essential to prove the equivalence of all the correct programs. Table 2, column “NI” shows the results of our ablation study when disabling automated functional induction. Results confirm that, compared to our base approach, this change only affects correct submissions – the percentage of programs classified is exactly the percentage of incorrect submissions.

Function Call Matching. Function call matching enables us to prove the equivalence of programs with auxiliary functions, as explained in Section 3.3. Table 2, column “NM” shows the results of our ablation study when disabling function call matching. Results confirm that function call matching is mandatory to prove the equivalence of correct submissions for the four benchmarks that contain auxiliary recursive functions (natmul, uniq, formula, and lambda).

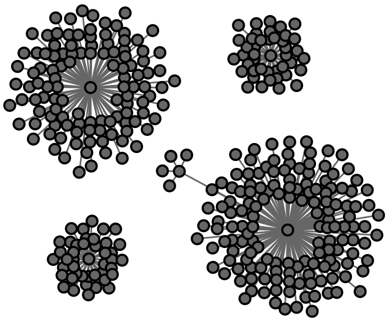

Fig. 5. Clusters of correct submissions found by our system for the *natadd* benchmark. Edges represent direct equivalence proofs. Clusters are formed around the four reference solutions.

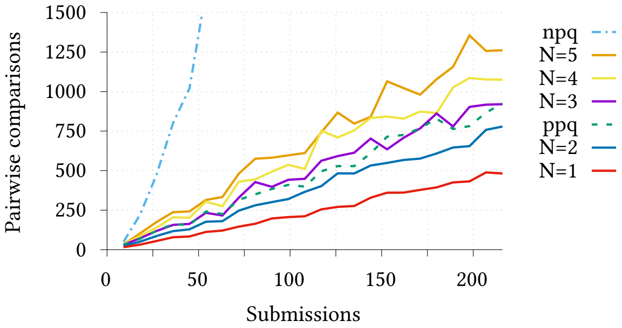

Fig. 6. Numbers of pairwise comparisons for the *max* benchmark. We consider 5 different values for the N parameter and compare our approach (N=3), with a variation that uses a simplified priority queue heuristic (ppq) and without priority queue (npq).

Multiple Reference Solutions. Our clustering algorithm allows instructors to provide more than one reference solution, that has, for example, a different recursion schema, or a different combination of auxiliary functions. Table 2, column "1RS" shows the results of our ablation study when only providing one reference solution. The use of additional reference solutions significantly improves the results for five benchmarks (sigma, natadd, natmul, uniq and lambda).

In addition to proving program correctness, our algorithm clusters correct submissions around the reference solution where it succeeds at proving the equivalence. Figure 5 shows such clusters for the natadd benchmark. The reference solutions are variations of each other, where addition is defined by recursion on the first or second argument, with or without tail recursion. Another example of multiple reference solutions with different recursion schemas is the sigma benchmark, where the main two reference solutions compute the sum $\sum_j^k f(i)$ starting from the opposite sides of the interval. A good example of multiple reference solutions with different auxiliary functions is the uniq benchmark, where one of the reference solutions builds the resulting list using an auxiliary function for list element removal, while another one builds the resulting list starting from an empty list, using an auxiliary function for adding unique elements.

5.6 Performance of Our Clustering Algorithm

In this section, we evaluate the performance of our clustering algorithm and consider the impact of two key components: the parameter N and the heuristic that we use to prioritize reference solutions. We additionally comment on the automatic discovery of intermediate reference solutions.

Impact of the N Parameter. To avoid the quadratic cost of checking the equivalence of all pairs of programs, our clustering algorithm only considers top N reference solutions in each iteration. N is a parameter of our algorithm, which we set to N=3 in our implementation. To illustrate the impact of our choice of N, Figure 6 shows the number of pairwise comparisons per number of input programs, for the 216 submissions from the max benchmark. We compare our implementation (N=3), with four other values for the N parameter (N=1, N=2, N=4, N=5). The graph shows 24 data points, each of which we obtained by running our algorithm 10 times and averaging the results. For each run, we took a random sample of the given size (in increments of 9). The graph suggests a linear dependence $y = kx$, where k is a function of N. For N=3, one run of the last iteration, which includes all 216 submissions, performs 920 pairwise comparisons. In terms of efficacy, all

```
def maxR(lst: List[Int]): Int =
  lst match
    case Nil() => −1
    case Cons(hd, Nil()) => hd
    case Cons(hd, tl) =>
      if (hd > maxR(tl)) hd
      else maxR(tl)
```

```
def maxC(l: List[Int]): Int =
  l match
    case Nil() => −1
    case Cons(a, Nil()) => a
    case Cons(a, Cons(b, tl)) =>
      if (a > b) maxC(a :: tl)
      else maxC(b :: tl)
```

```
def maxT(lst: List[Int]): Int =
  def bigger(a: Int, b: Int) =
    if (a >= b) a else b
  lst match
    case Nil() => −1
    case Cons(hd, tl) =>
      tl.foldLeft(hd)(bigger)
```

Fig. 7. The reference solution of the *max* benchmark (*maxR*) and two student submissions. Our system proves the equivalence of *maxT* and *maxR* by first proving the equivalence of *maxC* and *maxR*, and then proving the equivalence of *maxC* and *maxT*. The system automatically finds *maxC* as an intermediate reference solution, from the set of student submissions.

but the N=1 experiment succeed at proving the correctness of the same number of submissions (154). In our experiments, we find that N=3 works well even for benchmarks with more than three reference solutions, thanks to the priority queue used in our algorithm.

Impact of the Priority Queue. The priority queue of our algorithm plays a key role in the discovery of intermediate reference solutions, while avoiding the quadratic behavior. We assign priorities using a simple heuristic, which rewards proven equivalences and penalizes timeouts. Positive points ensure that reference solutions with higher success rates are considered first, thus reducing the number of timed-out comparisons. Negative points provide a basic round robin shuffling among new solutions. To illustrate the impact of our heuristic, Figure 6 additionally shows the number of pairwise comparisons when removing the priority queue and considering every reference solution (npq), and when using a simplified heuristic without negative points (ppq). Our heuristic performs better than the quadratic approach (npq), while still proving the same number of programs correct. The simplified heuristic (ppq) fails to discover the intermediate reference solution, which results in 15 less proven equivalences. Our heuristic is thus a clear win: it is as comparison-efficient as the simplified heuristic, while being as successful as the quadratic approach.

Intermediate Reference Solutions. Intermediate reference solutions appear when a student submission provides an induction schema that, for some other student submissions, works better than the one of initial reference solutions. While the idea of using intermediate programs to prove equivalence already appears in related work on regression verification [Felsing et al. 2014], what is notable is that our approach finds such programs automatically among other candidate submissions. Figure 7 shows the source code of the reference solution for the max benchmark (maxR), the discovered intermediate reference solution (maxC), as well as one of the 15 submissions that could not be proven equivalent to the reference solution directly (maxT). Note that our system has no built-in knowledge of functions such as foldLeft; it iteratively expands the definition of foldLeft from the list library and recovers sufficient approximate semantics for a sound correctness proof.

5.7 Comparison to REVE and RVT

In this section, we compare our system to two existing tools for automated equivalence checking of recursive programs: REVE [Felsing et al. 2014], and Regression Verification Tool (RVT) [Sayedoff and Strichman 2022; Strichman and Veitsman 2016]. We consider 17 benchmarks from the equivalence checking literature. Specifically, we combine the recursion benchmark suite from REVE (which includes some examples originally from RVT), as well as additional examples from recent

Table 3. Comparison between REVE [Felsing et al. 2014], RVT [Sayedoff and Strichman 2022; Strichman and Veitsman 2016], and our system. Checks (✓) indicate success in finding equivalence proofs, and crosses (✗) indicate failures. Benchmarks annotated with ★, □, and ● are from [Felsing et al. 2014], [Sayedoff and Strichman 2022], and [Strichman and Veitsman 2016], respectively.

	ackermann*	mccarthy91*	limit1*	limit2*	limit3*	add-horn*	triangular*	inlining*	sum$^\square$	fibonacci-f$^\square$	pascal$^\square$	fibonacci-m$^\square$	fibonacci-t$^\square$	fibonacci-h$^\square$	fact4$^\bullet$	fact13$^\bullet$	fact14$^\bullet$
REVE	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
RVT	✓	✓	✓	✗	✗	✓	✗	✗	✓	✓	✗	✗	✗	✓	✓	✗	✗
Our System	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

publications on RVT. Each benchmark consists of a pair of equivalent recursive programs. To evaluate REVE and RVT, we used the LLRÊVE web interface[1] and the Regression Verification Tool web interface[2]. We used the default settings for both tools. For REVE, we additionally tried all of the supported SMT solvers, and report results for the best outcome. For our system, we used Z3 with a 1 second timeout, both for termination and equivalence checking.

Table 3 summarizes the results we obtain by running the three tools on the 17 benchmarks. Both REVE and RVT prove the equivalence for 47% (8/17) of the benchmarks. Our system proves the equivalence for over 82% (14/17). This is remarkable because we are comparing on benchmarks given specifically by REVE and RVT to demonstrate the challenging examples where their tools succeed at proving the equivalence. While both tools perform poorly on each other's benchmark groups, our system yields good results for both groups. Furthermore, our system succeeds on all the benchmarks where original tools fail on their own examples, overcoming their limitations.

The three cases where our tool fails are the ackermann and mccarthy91 benchmarks, where it does not manage to automatically prove termination, as well as the triangular benchmark, where the two programs have auxiliary functions with different number of arguments. This is a typical example where our system would benefit from two reference solutions: one with a non-tail recursive function and the other with a tail-recursive auxiliary function with an additional argument. Our system proves the equivalence for all the remaining benchmarks, including programs where the original publication claims to be the only tool that, in its recent version, can automatically handle such cases (fibonacci−f and fibonacci−h).

5.8 Discussion

We conclude our evaluation with a discussion of our results in the context of the overarching goal of practical and rigorous automated grading.

Impact of the Z3 Timeout. For offline grading, an instructor might be tempted to specify arbitrarily large timeouts for SMT queries, which, in principle, can lead to better results. Conversely, for immediate feedback in a classroom setting, one might favor smaller timeouts. In our evaluation, we used a 1 second timeout for Z3. During our experiments, we noticed that larger timeouts do not lead to better results. One typical cause of failure, regardless of the Z3 timeout, comes from submissions that contain auxiliary recursive functions that are not equivalent to any of the operations from reference solutions. For example, a few submissions to the uniq assignment implement a function to reverse lists, an operation which does not exist in any of our reference solutions.

[1] https://formal.kastel.kit.edu/projects/improve/reve/index.php
[2] https://rvt.iem.technion.ac.il

Another typical cause of failure comes from our system's lack of knowledge about certain properties of primitive operations. For example, in order to prove the equivalence of the following two functions, our equivalence proof would need an additional lemma to establish associativity:

```
def find(lst: List[Int], n: Int): Boolean = lst match
  case Nil() => false
  case Cons(hd, tl) => (n == hd) || find(tl, n)
```

```
def isin(lst: List[Int], n: Int): Boolean =
  lst.foldLeft(false) { (acc, hd) =>
    (n == hd || acc) }
```

Automation and Instructor Interaction. Our approach is fully automatic and, by default, only takes as input a reference solution and a set of candidate submissions. It is fair to assume that instructors already wrote at least one correct reference solution for each problem. Writing multiple reference solutions helps to prove correctness of a diverse set of student submissions, but also requires certain additional effort. From a practical viewpoint, in the context of a programming course, solutions are naturally available for subsequent iterations.

Results of our evaluation indicate that our system manages to automatically prove or disprove equivalence for 86% of terminating submissions, leaving only a small percentage of assignments for further manual inspection. While not provably correct, unclassified programs did pass all the tests that we collect as counterexamples. By seeing those tests, instructors can gain a certain level of confidence regarding the correctness of unclassified programs, which helps with manual inspection. Refusing to classify a subset of programs is better than wrongly classifying them because it points to those programs that require manual analysis. In practice, we believe that some amount of manual inspection is acceptable, as long as there is only a low percentage of unclassified programs.

Not All Programming Courses Look Alike. In our evaluation, we used LearnML benchmarks to judge the applicability of our system on real functional programming assignments. These benchmarks are well suited for several reasons. They provide a large variety of recursive problems (15 assignments, 28 LOC on average), some of which involve auxiliary recursive functions (2 on average), and also have large numbers of submissions per assignment (311 on average). Furthermore, most submissions are correct (around 90%), which is important to accurately test the limits of our equivalence checking techniques.

We are aware that not all programming courses look alike and therefore our evaluation might not be representative of other programming courses. We mitigate this problem by evaluating our system on, to the best of our knowledge, the largest publicly available set of functional programming submissions, with over 4000 programs. We further address the concerns regarding the robustness of our techniques by evaluating our system on benchmarks from the equivalence checking literature. Results confirm that our techniques are suitable for a wide range of recursive problems.

Our implementation targets Scala programs, but our approach is applicable to other languages. In particular, functional induction is not Scala specific: it applies to recursive problems in general. Furthermore, our clustering algorithm and function call matching are entirely language agnostic.

6 RELATED WORK

There are various approaches to the automated grading problem. Our method relies on automated equivalence checking using proofs by induction. In this section, we describe related work on automated grading, equivalence checking and automated proofs by induction.

6.1 Automated Grading Tools

TestML [Song et al. 2019] is a tool for automatic detection of logical errors in OCaml programming assignments, combining enumerative search and symbolic execution. It only provides feedback in the form of counterexamples, and does not provide formal guarantee of program correctness. Furthermore, TestML is limited to analyzing programs individually, without making use of previously discovered counterexamples for subsequent submissions. In contrast, our technique makes use of clustering and is capable of reusing verification results and counterexamples.

OverCode [Glassman et al. 2015] is a system that aims at reducing the burden on instructors for large scale courses, by clustering similar solutions. Its main practical value lies in an interactive user interface for visualization of clusters of solutions, as well as manipulating and merging those clusters by adding rewrite rules. However, OverCode performs neither automated testing nor verification to check for program correctness, which remains a manual task for instructors.

EML [Singh et al. 2013] is a small error model language, designed for automated feedback generation in the context of introductory programming assignments in Python. The approach is automatic, but requires users to provide correct reference solutions, error models, as well as bounds for the input size, recursion depth and a limit for the number of loop unrollings. Clara [Gulwani et al. 2018] and Sarfgen [Wang et al. 2018] are tools for data-driven automated repair of introductory programming assignments, operating at the level of control flow graph. Another data-driven approach was proposed in sk_p [Pu et al. 2016], which uses deep learning techniques for program repair. Neither of these tools provides any formal guarantee of program correctness.

In [Vujosevic Janicic and Maric 2019], the authors present an approach for automated grading of imperative assignments, combining testing and bounded model checking. They rely on the verification tool LAV [Vujošević-Janičić and Kuncak 2012], that uses over-approximation and under-approximation to handle loops. Experimental evaluation demonstrates the practical value of their approach on 597 programs, most of which are not recursive and without auxiliary functions.

CoderAssist [Kaleeswaran et al. 2016] preforms clustering of C programs, specifically dynamic programming exercises, followed by teacher-guided verification within each cluster. It generates verified feedback for incorrect submissions, but requires significant manual effort to identify and write correct solutions within each cluster. Our approach also leaves a small percentage of unclassified programs for manual inspection, but provides counterexamples to help in the process.

LEGenT [Agarwal and Karkare 2022] is a tool for personalized feedback generation, using correct student submissions to find counterexamples in incorrect submissions. It uses Clara to split the submissions into clusters of semantically equivalent and structurally similar programs, and REVE to identify provably correct submissions. As the approach relies on Clara, it also requires manually written test cases. LEGenT does not currently handle recursion.

ZEUS [Clune et al. 2020] is a tool for clustering purely functional programs, not just by input/output behavior, but by structure. Unlike our technique, ZEUS does not aim at verifying program correctness and does not provide counterexamples for incorrect submissions. It also uses clustering of equivalent programs, and relies on transitivity to reduce the number of comparisons, but in the absence of reference solutions and without prioritizing solutions, this process still results in a large number of comparisons. Experimental results also indicate fairly large percentage of singleton clusters, which are left for manual inspection. When it comes to pairwise comparisons, ZEUS also relies on SMT solving, but while our technique uses functional induction, ZEUS uses inference rules that simulate relationships between expressions of the two programs. As a result, ZEUS is more restricted with respect to recursive programs and introduces limitations when programs use built-in Standard ML functions. Our work has no such limitations.

Nate [Seidel et al. 2017] is a tool for type-error localization in functional programming assignments. While this problem is orthogonal to our work, the benchmarks from Nate's evaluation are similar to ours (introductory OCaml programs, similar in size and programming constructs) [Seidel and Jhala 2017]. The data set contains over 5000 ill-typed programs and their fixes, many of which are identical, leaving under 2000 well-typed programs, after deduplication. Moreover, since Nate is only concerned with type errors, we found that the majority of well-typed programs still contain logical errors, and are thus of little use to evaluate our equivalence checking techniques.

6.2 Equivalence Checking Techniques

Our work relies on program equivalence for verifying program correctness, as it would be practically impossible to do so by writing correctness proofs for large numbers of student submissions. A similar idea is applicable in High Performance Computing, where optimizations are of great importance, and researchers have proposed benchmark sets in this area [Siegel and Zirkel 2011].

Regression verification is a method of proving that two versions of a program behave either equally or differently in a precisely specified way. This problem is often simpler than proving that a new version satisfies a complex functional specification. REVE [Felsing et al. 2014] exploits program similarity to automate equivalence proofs, using coupling predicates and weakest precondition calculus. One of the main limitations of their approach is that loops and function calls are always matched up in order of occurrence. One way to deal with complex examples is to intervene manually, by providing an intermediate implementation. In contrast, our algorithm naturally makes use of other candidate submissions as intermediate implementations, thus removing the need for manual intervention. Our technique makes use of structural similarity only at the level of matching function calls; it additionally performs function inlining during inductive proofs, which support differences in the decomposition of programs into functions between equivalent solutions.

RVT [Strichman and Godlin 2005] is another tool for automated regression verification, with recent focus on programs with unbalanced recursive functions [Sayedoff and Strichman 2022; Strichman and Veitsman 2016]. RVT is designed for comparison of large programs, and relies on a decomposition algorithm based on abstracting equivalent functions with uninterpreted functions [Godlin and Strichman 2013]. The underlying matching algorithm looks for functions with the same name, return type, and prototype, and also takes into account mappings of global variables. Our matching algorithm improves on this strategy, notably by using testing and considering permutations of function arguments. Unlike our system, RVT does not aim at disproving equivalence: it does not differentiate non-equivalent programs from programs that the tool fails to prove equivalent.

In [Fuhs et al. 2017], the authors propose the use of logically constrained term rewriting systems for equivalence checking of procedural programs. To prove properties of recursive definitions, they use induction on term reduction. Their technique is fully automatic and integrates termination checking and equivalence checking, similarly to ours. The authors mention that their technique is applicable to automated grading and evaluate it on several examples. However, they do not assess the practicality of their tool for this particular use case. The authors also mention that disproving could be used to extract counterexamples, but currently do not provide such functionality.

Other examples of techniques that aim to explore possibilities of automating equivalence proofs by exploiting program similarity are presented in [Lahiri et al. 2012], which uses intermediate verification language Boogie in order to provide support for several imperative languages, and [Lahiri et al. 2013], which compares program versions to ensure that certain properties are preserved across program changes. Prior work on cross-version program analysis includes particular instances of regression verification, targeting version control systems [Sousa et al. 2018], as well as focusing on equivalence in the presence of memory allocations [Wood et al. 2017].

Particular applications of program equivalence also include verifying loop optimizations, using simulation relations [Sharma et al. 2013], or product programs [Churchill et al. 2019]. Translation validation [Dahiya and Bansal 2017; Kundu et al. 2009; Lopes et al. 2021; Zuck et al. 2005] is an example of equivalence checking for verifying compiler correctness by validating each individual translation. This problem is less general than program equivalence, which makes the validation of individual translations potentially easier than proving compiler correctness for all programs. Much of the related work for imperative programs applies to programs that do not have a rich theory of algebraic data types or the diversity of recursion exhibited as in our data set. Furthermore, the use cases of translation validation do not include clustering of many versions of the same program.

Nebula [Kolesar et al. 2022] is a fully automated expression equivalence checker for programs in a non-strict functional language, using symbolic execution and coinduction. To evaluate Nebula, the authors consider benchmarks from inductive theorem provers [Sonnex et al. 2012]. In contrast, our evaluation focuses on a larger set of functional programming assignments.

6.3 Automated Proofs by Induction

Induction is widely used in automated theorem provers. ACL2 [Kaufmann et al. 2000] uses sophisticated heuristics to automatically decide which induction schemas to use. Coq [Bertot and Castéran 2004] implements functional induction as a tactic that performs case analysis and induction using a definition of the function specifying the induction principle [INRIA 2021]. Isabelle [Nipkow et al. 2002] offers great flexibility when it comes to manually specifying induction schemas and has recent advances in automating induction in the form of a recommendation tool [Nagashima 2020].

SMT solvers traditionally do not natively support inductive reasoning and do not make completeness claims with respect to deriving consequences of recursive function definitions. In [Leino 2012], the author proposes preprocessing of formulas in the form of a tactic that identifies properties whose proofs may benefit from induction, and describes its implementation in the Dafny program verifier. Researchers have also introduced inductive reasoning into the theorem prover Vampire [Reger and Voronkov 2019], obtaining competitive results on benchmarks. CVC4 contains a built-in approach to induction [Reynolds and Kuncak 2015], activated with a command-line option. One of the advantages of CVC4's approach is the ability to exploit the internal state of the solver to discover subgoals that themselves require proofs by induction. The paper also explores automatic subgoal discovery by enumerating potential equalities.

Researchers have built dedicated tools for automating inductive proofs [Johansson et al. 2006; Sonnex et al. 2012]. HipSpec [Claessen et al. 2012], Hipster [Johansson et al. 2014] and TheSy [Singher and Itzhaky 2021] are capable of introducing auxiliary lemmas for automated inductive proofs, using theory exploration. They employ structural induction, enumerating possible candidates. In contrast, our technique directly considers proofs by functional induction. Both HipSpec and Hipster apply filtering techniques based on testing, using QuickSpec [Claessen et al. 2010]. We use counterexamples found by our system, which does not require writing generators and uses SMT solvers instead. Our system only looks for auxiliary lemmas corresponding to equivalence of auxiliary functions. Complementing our approach further with theory exploration could be beneficial in our context, for programs that result in a timeout due to missing lemmas.

Many automated induction provers do not make use of recursive function definitions as a source of induction schemas. Doing so requires the prover to first establish that recursive definitions are well founded, which ACL2 and our system perform as a dedicated task. Coq and Isabelle rely on a mix of built-in tactics for common cases and user-specified measures. Whereas it would be possible in principle to manipulate Scala programs in proof assistants, our experience suggests that it would be difficult to do so robustly and automatically on a large and diverse data set such as ours.

ACKNOWLEDGMENTS

This work is supported in part by the Swiss National Science Foundation Project number 200021_197288. We thank our shepherd Sam Tobin-Hochstadt and the anonymous reviewers for their thoughtful and helpful comments and suggestions. The authors would also like to thank Julie Giunta for the translation of OCaml benchmarks, Mario Bucev for advice on implementation aspects, as well as Simon Guilloud, Georg Schmid, Olivier Blanvillain, and Jad Hamza for their feedback and help on this work.

AVAILABILITY OF DATA AND SOFTWARE

Stainless is under active development and is available at [EPFL 2023]. The complete data set and instructions for reproducing the results from this paper are available in open access Zenodo repository [Milovancevic and Kuncak 2023].

REFERENCES

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

Rajeev Alur and Pavol Černý. 2011. Streaming Transducers for Algorithmic Verification of Single-Pass List-Processing Programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 599–610. https://doi.org/10.1145/1926385.1926454

Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3368089.3409757

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Yves Bertot and Pierre Castéran. 2004. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* https://doi.org/10.1007/978-3-662-07964-5

Tej Chajed, Haogang Chen, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich, and Daniel Ziegler. 2017. Certifying a File System Using Crash Hoare Logic: Correctness in the Presence of Crashes. *Commun. ACM* 60, 4 (mar 2017), 75–84. https://doi.org/10.1145/3051092

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2012. HipSpec: Automating Inductive Proofs of Program Properties. In *ATx'12/WInG'12: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation, Manchester, UK, June 2012 (EPiC Series in Computing, Vol. 17)*, Jacques D. Fleuriot, Peter Höfner, Annabelle McIver, and Alan Smaill (Eds.). EasyChair, 16–25. https://doi.org/10.29007/3qwr

Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. 6–21. https://doi.org/10.1007/978-3-642-13977-2_3

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

Manjeet Dahiya and Sorav Bansal. 2017. Black-Box Equivalence Checking Across Compiler Optimizations. 127–147. https://doi.org/10.1007/978-3-319-71237-6_7

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

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

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

Carsten Fuhs, Cynthia Kop, and Naoki Nishida. 2017. Verifying Procedural Programs via Constrained Rewriting Induction. *ACM Trans. Comput. Logic* 18, 2, Article 14 (jun 2017), 50 pages. https://doi.org/10.1145/3060143

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

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

Simon Guilloud, Mario Bucev, Dragana Milovancevic, and Viktor Kuncak. 2023. Formula Normalizations in Verification. (2023), 18. http://infoscience.epfl.ch/record/297701

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

Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-Guided Correlation Algorithm for Translation Validation. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 221 (nov 2020), 29 pages. https://doi.org/10.1145/3428289

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

Gernot Heiser, Gerwin Klein, and June Andronick. 2020. seL4 in Australia: from research to real-world trustworthy systems. *Commun. ACM* 63, 4 (2020), 72–75. https://doi.org/10.1145/3378426

INRIA. 2007a. Batch compilation (ocamlc). https://ocaml.org/manual/comp.html.

INRIA. 2007b. Camlp5. https://github.com/camlp5/camlp5.

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

Moa Johansson, Alan Bundy, and Lucas Dixon. 2006. Best-First Rippling. In *Reasoning, Action and Interaction in AI Theories and Systems, Essays Dedicated to Luigia Carlucci Aiello (Lecture Notes in Computer Science, Vol. 4155)*, Oliviero Stock and Marco Schaerf (Eds.). Springer, 83–100. https://doi.org/10.1007/11829263_5

Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. 2014. Hipster: Integrating Theory Exploration in a Proof Assistant. In *Intelligent Computer Mathematics*, Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban (Eds.). Springer International Publishing, Cham, 108–122. https://doi.org/10.1007/978-3-319-08434-3_9

Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-Supervised Verified Feedback Generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 739–750. https://doi.org/10.1145/2950290.2950363

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

Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. 2018. Formally Verified Software in the Real World. *Commun. ACM* 61, 10 (sep 2018), 68–77. https://doi.org/10.1145/3230627

John C. Kolesar, Ruzica Piskac, and William T. Hallahan. 2022. Checking Equivalence in a Non-Strict Language. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 177 (oct 2022), 28 pages. https://doi.org/10.1145/3563340

Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. *SIGPLAN Not.* 44, 6 (June 2009), 327–337. https://doi.org/10.1145/1543135.1542513

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. https://doi.org/10.1007/978-3-642-31424-7_54

Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 345–355. https://doi.org/10.1145/2491411.2491452

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

K. Rustan M. Leino. 2012. Automating Induction with an SMT Solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation* (Philadelphia, PA) *(VMCAI'12)*. Springer-Verlag, Berlin, Heidelberg, 315–331. https://doi.org/10.1007/978-3-642-27940-9_21

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

John McCarthy. 1963. A Basis for a Mathematical Theory of Computation1). 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. https://doi.org/10.1016/S0049-237X(08)72018-4

Dragana Milovancevic and Viktor Kuncak. 2023. *Proving and Disproving Equivalence of Functional Programming Assignments (Artifact)*. https://doi.org/10.5281/zenodo.7712002

Yutaka Nagashima. 2020. Smart Induction for Isabelle/HOL (Tool Paper). In *2020 Formal Methods in Computer Aided Design (FMCAD)*. 245–254. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_32

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

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

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

Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_p: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (Amsterdam, Netherlands) *(SPLASH Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 39–40. https://doi.org/10.1145/2984043.2989222

Giles Reger and Andrei Voronkov. 2019. Induction in Saturation-Based Proof Search. In *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11716)*, Pascal Fontaine (Ed.). Springer, 477–494. https://doi.org/10.1007/978-3-030-29436-6_28

Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–98. https://doi.org/10.1007/978-3-662-46081-8_5

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

Chaked R. J. Sayedoff and Ofer Strichman. 2022. Regression verification of unbalanced recursive functions with multiple calls (long version). https://doi.org/10.48550/ARXIV.2207.14364

Eric L Seidel and Ranjit Jhala. 2017. A Collection of Novice Interactions with the OCaml Top-Level System. https://doi.org/10.5281/zenodo.806814

Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 60 (oct 2017), 27 pages. https://doi.org/10.1145/3138818

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-Driven Equivalence Checking. *SIGPLAN Not.* 48, 10 (Oct. 2013), 391–406. https://doi.org/10.1145/2544173.2509509

Stephen F. Siegel and Timothy K. Zirkel. 2011. FEVS: A Functional Equivalence Verification Suite for High-Performance Scientific Computing. *Mathematics in Computer Science* 5, 4 (01 Dec 2011), 427–435. https://doi.org/10.1007/s11786-011-0101-6

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

Eytan Singher and Shachar Itzhaky. 2021. Theory Exploration Powered by Deductive Synthesis. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*. Springer-Verlag, Berlin, Heidelberg, 125–148. https://doi.org/10.1007/978-3-030-81688-9_6

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

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

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7214)*, Cormac Flanagan and Barbara König (Eds.). Springer, 407–421. https://doi.org/10.1007/978-3-642-28756-5_28

Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified Three-Way Program Merge. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 165 (oct 2018), 29 pages. https://doi.org/10.1145/3276535

Ofer Strichman and Benny Godlin. 2005. Regression Verification - A Practical Way to Verify Programs. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions (Lecture Notes in Computer Science, Vol. 4171)*, Bertrand Meyer and Jim Woodcock (Eds.). Springer, 496–501. https://doi.org/10.1007/978-3-540-69149-5_54

Ofer Strichman and Maor Veitsman. 2016. Regression Verification for Unbalanced Recursive Functions. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International

Publishing, Cham, 645–658. https://doi.org/10.1007/978-3-319-48989-6_39

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

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

Nicolas Charles Yves Voirol. 2019. Verified Functional Programming. (2019), 229. https://doi.org/10.5075/epfl-thesis-9479

Milena Vujošević-Janičić and Viktor Kuncak. 2012. Development and Evaluation of LAV: An SMT-Based Error Finding Platform. In *Verified Software: Theories, Tools, Experiments*, Rajeev Joshi, Peter Müller, and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–113. https://doi.org/10.1007/978-3-642-27705-4_9

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

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

Tim Wood, Sophia Drossopolou, Shuvendu K. Lahiri, and Susan Eisenbach. 2017. Modular Verification of Procedure Equivalence in the Presence of Memory Allocation. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings* (Uppsala, Sweden). Springer-Verlag, Berlin, Heidelberg, 937–963. https://doi.org/10.1007/978-3-662-54434-1_35

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

Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. 2005. Translation and Run-Time Validation of Loop Transformations. *Formal Methods in System Design* 27 (11 2005), 335–360. https://doi.org/10.1007/s10703-005-3402-z

Received 2022-11-10; accepted 2023-03-31