Clustering Framework for Scala Programming Assignments Based on Stainless

Shizhe Cui

Master of Science
School of Informatics
University of Edinburgh
2024

Abstract

Manually grading programming assignments is a labour-intensive task. The numerous repetitive errors in these assignments require human graders to spend considerable time and effort on redundant tasks. Clustering equivalent assignments is an effective solution to this problem. We explored the use of Scala program equivalence proofs with the Stainless formal verifier for clustering Scala programming assignments in the Elements of Programming Languages course. To overcome Stainless's limitations in supporting standard Scala and its inability to verify complex exercises, we proposed an automated transformation method for the original programming assignments and implemented a Stainless-based clustering framework. In the 2023 EPL programming assignment submissions, which included 27 exercises, our prototype successfully transformed an average of 96% of submissions per exercise into a Scala subset accepted by Stainless. Additionally, it effectively reduced the number of unknown equivalence checks in 6 exercises through optimization strategies and got sound clustering results in all exercises.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor James Cheney. His profound expertise in formal verification has been invaluable to my research, and his rigorous academic attitude have greatly influenced my development. I am truly honored to have had the opportunity to work under his guidance.

Next, I would like to extend my heartfelt thanks to my parents and family. Throughout my academic journey, my parents have provided unwavering financial and emotional support, doing everything in their power to create the best possible conditions for my success. Words cannot fully capture the depth of my love and appreciation for them. I am also deeply grateful to my aunt, Shujie Liu, for her care, and to my cousin, Peiyi Chen, for her companionship and encouragement. I am truly fortunate to have them as my family.

I would also like to express my sincere thanks to my friends in Edinburgh. I am especially grateful to Wenjia Geng and Yijie Xu, my fellow classmates in the program, who made me feel less alone in a foreign country. It was a true pleasure to share this year-long academic journey with them. Lastly, I want to express my heartfelt gratitude to Kaiting Mai. She passed through my life like a beautiful shooting star—brief but brilliant enough to leave a lasting impression that I will cherish forever.

Table of Contents

Chapter 1

Introduction

Manually grading and providing feedback on programming assignments is a labour-intensive task. Students often make similar errors in their assignments, and manually checking each one demands considerable time and effort from human graders on repetitive tasks. Additionally, manually reading code is also prone to errors.

Testing is the most common solution to this issue [1]. However, many test cases are required to ensure program coverage and produce accurate results [2], which may also incur additional manual costs. Another approach is to cluster programming assignments based on program equivalence or similarity. This allows human graders to provide a unified score and feedback for the entire cluster. Previous research has achieved this functionality in various programming languages using a range of approaches, including program analysis, symbolic execution, model checking, and theorem proving [3]–[9].

The primary goal of this project is to achieve clustering of Scala programming assignments 1 and 2 in the Elements of Programming Languages (EPL) course[1] based on equivalence checking provided by Stainless formal verifier [8]. Stainless provides semantic-based program equivalence checking, offering results with formal guarantees. Although it has been previously used for clustering Scala programming assignments [9], there has been no specific research focusing on assignments like EPL programming tasks, which directly utilize standard Scala and contain relatively complex exercises. To achieve this, we translated the original standard Scala assignments and applied specific optimizations to reduce the difficulty of verification with Stainless through automated transformations. In the 2023 EPL programming assignment submissions containing 27 exercises, our method enabled an average of 96% of submissions per exercise to be successfully converted into a Scala version accepted by Stainless. In 6 of the exercises,

[1]`https://www.inf.ed.ac.uk/teaching/courses/epl`

the application of optimization strategies successfully reduced the number of unknown equivalence-checking results during the clustering process. And the final clustering results are all sound.

The main contributions of this dissertation are:

- Stainless Support for EPL assignments: We enabled Stainless to accept and process EPL programming assignments by transformation, which will be discussed in Chapter 3. This includes translating standard Scala to Stainless Scala and a targeted extraction method that extracts specific exercises to prevent interference from others.

- Performance Optimization: In Chapter 4, we proposed several strategies to optimize performance for some exercises, thereby reducing the difficulty of equivalence checking in Stainless and minimizing the occurrence of timeouts.

- Clustering: We extended the existing clustering algorithm in Stainless to support the clustering of student submissions with the same semantics in Chapter 5, enhancing its practicality in real educational settings.

- Implementation and Evaluation: We implemented our approach based on Stainless and evaluated it on real EPL programming assignment submissions in 2023. The evaluation results on the real dataset demonstrate the effectiveness of our approach. These will be discussed in Chapter 6 and Chapter 7.

Chapter 2

Background

This chapter first introduces the working principles and capabilities of the Stainless verifier for Scala program equivalence checking in Section 2.1. In Section 2.2, we describe the content and format of the EPL course programming assignments that need to be clustered, along with the issues that need to be addressed.

2.1 Stainless Formal Verifier

Stainless is a Scala formal verification tool. It can statically verify whether a Scala program terminates and whether it meets program specifications, which can be either automatically generated or provided by the developer. It relies on the higher-order functional language solver Inox [10]. Stainless will translate the Scala Abstract Syntax Tree (AST) into the AST of a pure functional verification language supported by Inox. The latter ultimately embeds the verification language into quantifier-free Satisfiability Modulo Theories (SMT) expressions and uses SMT solvers like Z3 [11] and CVC5 [12] for solving. The design of Stainless is similar to a combination of a bounded model checker and a k-inductive prover [13], which can provide proof when the specifications hold and generate corresponding counterexamples when they do not.

Stainless supports equivalence checking between Scala programs [8]. By providing two Scala functions with the same signature, it can automatically construct the corresponding postconditions to verify whether their outputs are equivalent for all inputs. If they are not equivalent, it outputs a set of parameters as counterexamples. For programs with recursive calls, Stainless uses functional induction to prove their equivalence. Stainless's equivalence checking also includes some more advanced features. When Stainless cannot prove that two programs are equivalent and no counterexample is found, it can search for function calls with matching parameters and attempt to first prove the equivalence of these two auxiliary functions. If they are found to be equivalent, Stainless will replace the call to one auxiliary function with the other, thereby reducing the complexity of the original proof. Additionally, it includes a clustering feature that can classify a set of candidates as correct, incorrect, or unknown according to the reference models.

2.2 Program Assignments in EPL course

The exercises included in EPL Programming Assignments 1 and 2 and their descriptions are shown in Table 2.1 and Table 2.2. Assignment 1 primarily consists of warm-up exercises focusing on features of the Scala language. Assignment 2 is more complex, requiring students to implement an interpreter and a type checker, as well as capture-avoiding substitution and desugaring for a simple programming language called Giraffe. Both these two assignments are structured as Scala files, with each student submitting an individual Scala file containing the provided framework code and some definitions that students need to complete for each exercise. Students are also permitted to add auxiliary functions or other definitions to complete the exercises.

EPL programming assignments 1 and 2 involve using various Scala language features, including common data types like Int and String, type parameters, higher-order functions, lambda functions and data structures such as List and Map. Most of these features are supported by Stainless [14]. However, Stainless does not directly support the standard Scala used in EPL programming assignments. It is limited to analyzing its own implementation of the Stainless library and cannot analyze the Scala standard library. An automated method is required to transform the original Scala code to make it compatible with Stainless. This process will be discussed in Chapter 3. Some of the complex exercises may pose challenges for Stainless in equivalence checking, leading to unknown results. Some performance optimization strategies are presented to reduce the verification complexity in Chapter 4. Additionally, a new clustering algorithm is needed to group assignments with equivalent semantics based on the results of Stainless's equivalence checks, which will be discussed in Chapter 5.

No	Name	Description
1	polynomial	Compute the value of a polynomial expression
2	sum	Sum all numbers from 0 to n
3	cycle	Rotate the components of a triple
4	suffix	Compute the suffix of a given number
5	boundingBox	Compute the smallest rectangle that encloses all points
6	mayOverlap	Check if two bounding boxes overlap
7	compose1	Compose two given functions
8	compose	Combine two functions into a single function
9	map	Apply a given function to each element in a list
10	filter	Filter a list based on a given predicate
11	reverse	Reverse the order of elements in a list
12	lookup	Look up a value in a key-value pair list
13	update	Update the value associated with a key in a list
14	keys	Return all keys present in a key-value pair list
15	presidentListMap	Map numbers to the names of U.S. presidents
16	map12	Define a map and update it with additional entries
17	list2map	Convert a list of pairs into a map
18	election	Count the occurrences of each element in a list

Table 2.1: Description of EPL Programming Assignment 1

No	Name	Description
1	Value.multiply	Perform multiplication operation on Giraffe numbers
2	Value.eq	Test for equality between Giraffe values
3	Value.length	Calculate the length of a Giraffe string
4	Value.index	Access a specific index within a Giraffe string
5	Value.concat	Concatenate two Giraffe strings into one
6	eval	Interpret the Giraffe programming language
7	tyOf	Perform type checking on Giraffe expressions
8	subst	Execute capture-avoiding substitution within Giraffe expressions
9	desugar	Simplify Giraffe expressions by removing syntactic sugar

Table 2.2: Description of EPL Programming Assignment 2

Chapter 3

Assignments Transformation

The standard Scala used in EPL programming assignments is not fully compatible with Stainless, making it impossible to directly process these assignments with Stainless. To resolve this issue, this chapter introduces a set of translation rules for converting standard Scala into Stainless Scala, allowing for the automated transformation of EPL programming assignments while ensuring the maximum preservation of their semantics. Additionally, a target extraction algorithm is proposed to isolate specific exercises within EPL programming assignments, ensuring that unsupported exercises do not interfere with other exercises.

3.1 Translation for Stainless Scala

3.1.1 General Type Translation

Stainless cannot directly process types from the Scala library. Instead, it defines its own library called Stainless library, which implements types corresponding to some of those in the Scala standard library [15]. To translate the original Scala code into a subset of Scala supported by Stainless, we need to replace the original types with semantically similar Stainless types. The predefined types in Stainless include Int, BigInt, String, Real, Set, Array, Map, Tuple and Bitvectors. Except for Real and BitVectors, these types correspond to those with the same names in Scala's scala package and Predef package. Real corresponds to scala.math.BigDecimal, while BitVectors do not have a corresponding type in Scala. These types are directly supported by Stainless's backend solver, Inox. Additionally, Stainless defines some common data structures in its library as ordinary algebraic data types (ADTs), such as Option, List, and ListMap, which correspond to scala.Option, scala.collection.immutable.List, and scala.collection.immutable.ListMap, respectively.

```
/* Before */
def map12_withUpdate: scala.collection.immutable.ListMap[Int, String
    ] =
{
  val map12 = scala.collection.immutable.ListMap[Int, String]();
  map12 + (1 -> "a", 2 -> "b");
}

/* After */
def map12_withUpdate: stainless.collection.ListMap[OverflowInt,
    String] =
{
  val map12 = stainless.collection.ListMap[OverflowInt, String]();
  map12 + (OverflowInt(1) -> "a") + (OverflowInt(2) -> "b");
}
```

Listing 3.1: An example of type translation.

The translation of types relies on replacing type constraints and constructors, with additional constructors being added in certain cases. Listing 3.1 shows an example. The first map12_withUpdate represents the original Scala source code, while the second map12_withUpdate represents the Stainless Scala source code that can be automatically generated after applying the translation rules. In the translated code, the scala.collection.immutable.ListMap type and its constructor calls have been replaced with the equivalent stainless.collection.ListMap. Some additional transformation rules are also applied to address the subtle differences between the Scala and Stainless types. The original `map12 + (1 -> "a", 2 -> "b")` expression was split into two separate + calls. This is because methods not directly supported by Inox cannot utilize varargs in Stainless, meaning that the + method in Stainless ListMap can only accept one key-value pair at a time.

Listing 3.1 also illustrates a relatively specific general translation: we use OverflowInt instead of Stainless Int and add the OverflowInt constructor for integer literals for conversion. The consideration is compatibility with other APIs in the Stainless library and integer overflow. Some interfaces of other Stainless types only accept or return BigInt types, and Stainless only allows converting Int literals to BigInt, not permitting any other conversions between them. This means that an original Scala program using Int may result in type conversion errors after translation from Scala Int to Stainless Int. If we choose to translate Scala Int to Stainless BigInt, it will result in semantic differences between the original program with integer overflow and the translated program, potentially leading to unsound results [16].

```
@library
case class OverflowInt(underlying: BigInt) {
  require(underlying >= OverflowInt.IntMin && underlying <=
     OverflowInt.IntMax, "OverflowInt out of bounds")

  def +(that: OverflowInt): OverflowInt = OverflowInt.overflow(
     underlying + that.underlying)
  ...
}

@library
object OverflowInt {
  ...
  private def overflow(num: BigInt): OverflowInt = {
    if (num > IntMax) OverflowInt(num - IntMax - 1 + IntMin)
    else if (num < IntMin) OverflowInt(num - IntMin + 1 + IntMax)
    else OverflowInt(num)
  }
  ...
}
```

Listing 3.2: The Definition of OverflowInt

To address this issue, we introduced a new BigInt wrapper class called OverflowInt in the Stainless library as an equivalent replacement for Int. This class uses BigInt to simulate the integer overflow rules of Scala's Int, wrapping around on overflow and underflow to match the semantics of the original Scala Int as shown in Listing 3.2. Therefore, it can ensure consistency with the original semantics of Int. Additionally, it can be converted to and from BigInt, ensuring compatibility with other interfaces in Stainless.

```
/* Before */
case _ if 11 until 13 contains n % 100 => "th"

/* After */
case _ if stainless.collection.List.range(OverflowInt(11),
    OverflowInt(13)) contains n % OverflowInt(100) => "th"
```

Listing 3.3: Translating until to List.range.

For some types that are not supported by Stainless, we can substitute them with other types. One example is shown in Listing 3.3. While Stainless does not support ranges created by the until keyword, they can be substituted with a Stainless List containing the equivalent integers. We will translate Scala's to and until keywords into calls to the stainless.collection.List.range method, which generates a list of integers within the specified range.

3.1.2 Specialized Type Translation

```
// Provided framework code
abstract class Shape
case class Circle(r: Double, x: Double, y: Double) extends Shape
case class Rectangle(llx: Double, lly: Double, w:Double, h:Double)
    extends Shape

// The exercises students need to complete
def boundingBox(s: Shape): Rectangle = s match {
    case Rectangle(lrx, lry, w, h) => Rectangle(lrx, lry, w, h)
    case Circle(r, x, y) => Rectangle(x - r, y - r, 2 * r, 2 * r)
}
```

Listing 3.4: The boundingBox exercise involving the Double type

Exercises boundingBox and mayOverlap in the EPL programming assignment 1 include the Double type, which is not supported by Stainless. However, this usage is limited to the provided framework code and does not involve Double literals as shown in Listing 3.4. Students are not required to introduce new Double variables or use Double literals to complete these exercises. In these cases, we will translate Scala's Double to Stainless's BigInt. The reason for choosing BigInt is that Double here represents coordinates in a Cartesian coordinate system, and there is no size limit to them in semantics. For other exercises where the Double type appears in students' code, we will reject these because translating Double to another type is currently unsafe.

3.1.3 Exception

In EPL programming assignments, there is no need to use the try-catch exception handling mechanism that Stainless does not support. However, the assignments may include calls to scala.sys.error or direct throwing of exceptions. Stainless provides stainless.lang.error, which is directly supported by the Inox solver and offers functionality similar to scala.sys.error. It can also serve as an alternative to throwing exceptions. However, stainless.lang.error presents additional issues in program equivalence checking. The original design of Stainless's equivalence checking did not take into account programs that include calls to stainless.lang.error [8]. By default, the preliminary verification in Stainless will reject programs that may execute stainless.lang.error, as it is considered unsafe behaviour akin to throwing exceptions. Even if this verification is disabled, equivalence checking will treat these programs as non-equivalent, even if they call stainless.lang.error under the same conditions with the same arguments. This is because different stainless.lang.error calls with the same arguments are translated into calls to different uninterpreted functions in the final SMT formula by Inox, and such function calls are not considered equivalent.

```scala
/* Package 1 */
def lookup[K, V](m: List[(K, V)], k: K): V = m match {
        case Nil => sys.error("Not found.")
        case (a, b) :: t if a == k => b
        case h :: t => lookup(t, k)
}
/* Package 2 */
def lookup[K, V](m: List[(K, V)], k: K): V = m match {
        case Nil => throw new Exception("Cant find the value.")
        case (m,n) :: xs if m == k => n
        case (m,n) :: xs => lookup(xs, k)
}
```

Listing 3.5: Two equivalent functions that throw exceptions in different ways.

This imposes certain limitations on Stainless's equivalence checking, especially in the context of verifying programming assignments. Listing 3.5 shows an example where two versions of lookup throw exceptions with different methods and messages when m is Nil. These two functions should be considered equivalent because they both terminate with an exception under the same conditions, and their behaviour is also equivalent for other inputs. However, if we simply translate sys.error and the throwing of exceptions into calls to stainless.lang.error, Stainless will not be able to handle them correctly.

To enable Stainless to correctly support this kind of equivalence checking with exceptions, we defined a wrapper function for sys.error in the Stainless library called errorWrapper and translated all calls to sys.error and the throwing of exceptions into calls to it. Its definition and translation result is shown in Listing 3.6. It has no parameters, so the original exception message will be ignored. @extern is an annotation in Stainless used to ignore the function body and only extract its contracts. This annotation ensures that errorWrapper is translated into an uninterpreted function that returns Nothing in the SMT formula, making all calls to it equivalent. Due to Stainless's special handling of the Nothing type, some modifications to Stainless are required for errorWrapper to function correctly. This will be discussed in the Section 6.4.

```
@extern def errorWrapper: Nothing = sys.error("error message")

/* Before */
case Nil => sys.error("Not found.")
case Nil => throw new Exception("Cant find the value.")

/* After */
case Nil => errorWrapper
case Nil => errorWrapper
```

Listing 3.6: Definition and usage of errorWrapper

```
/* Before */
n % 10 match {
  case 1 => "st"
}

/* After */
n % OverflowInt(10) match {
  case OverflowInt(1) => "st"
  case _ => errorWrapper
}
```

Listing 3.7: Non-exhaustive-match

```
/* Before */
// env is a Map
case Var(x) => {
    env(x)
}

/* After */
case Var(x) => {
    env.getOrElse(errorWrapper)
}
```

Listing 3.8: Translation for Map::Apply

Based on errorWrapper, we can further extend the scope of Stainless equivalence checking to support programs that include implicit exception throwing. By default, Stainless refuses to perform equivalence checking on programs that do not satisfy match exhaustiveness because they contain an implicit default branch that throws an exception. For such pattern matches, we will automatically add a default branch that throws an exception using errorWrapper as shown in Listing 3.7. Similarly, for methods like Map::apply that may potentially throw exceptions, we also transform them to throw exceptions using errorWrapper explicitly. A typical example is shown in Listing 3.8.

3.2 Target Extraction

The translation of programming assignments cannot guarantee that the resulting programs will be compatible with Stainless, as we have not restricted the language features students may use. Stainless will check all input Scala source code for unsupported features before verification. When unsupported features are present, Stainless will throw an error and exit. This can cause some normal exercises in EPL programming assignments to be rejected by Stainless. In both assignment 1 and assignment 2 of the EPL course, all of a student's submitted exercises are contained within a single Scala file. If one of the exercises contains unsupported syntactic features, all of the student's exercises will be rejected by Stainless, even if we are verifying a valid exercise. Listing 3.9 shows an example. The reverse exercise cannot be processed by Stainless because it prohibits modifications to mutable variables within the lambda function passed to the foreach method. When we attempt to perform an equivalence check between the election and its other implementation, it will be rejected because it shares the same source file as reverse, even though the election and reverse exercises are unrelated.

```scala
/* One source file submmited by a student */
// Unsupported exercise
def reverse[A](l: List[A]): List[A] = {
  var outputList: List[A] = Nil()
  // Stainless doesn't support effects in lambda body
  l.foreach { i =>
    outputList = i :: outputList
  }
  outputList
}

// Helper methods added by students
def addVote(l: List[String], m: ListMap[String, Int]): ListMap[
   String, Int] =
  ...

// Another exercise
def election(votes: List[String]): ListMap[String, Int] =
  val listmap = ListMap[String, Int]()
  addVote(votes, listmap)
```

Listing 3.9: Examples of Unsupported Features and Multi-function Exercises

Algorithm 1: Extract Specific Targets and Dependencies

```
Input: Extract targets (targets)
Output: Extracted definitions (result)
1  Function AddResult(item):
2      if $item \notin result$ then
3          $result \leftarrow result \cup item$
4          $worklist \leftarrow worklist \cup item$
5  $result \leftarrow targets$
6  $worklist \leftarrow targets$
7  while $worklist \neq \emptyset$ do
8      remove definition from worklist
9      if definition is class T ... or is trait T ... then
10         foreach parentType in GetParentTypes(T) do
11             AddResult(parentType)
12         foreach subType in GetSubTypes(T) do
13             AddResult(subType)
14     else if definition is companion object O ... then
15         foreach type in GetType(O) do
16             AddResult(type)
17     foreach expression in GetExpressions(definition) do
18         if expression is $F(\ldots)$ then
19             AddResult($F$)
20         else if expression is $x : T$ or new $T(\ldots)$ then
21             AddResult($T$)
22         else if expression is O and O is companion object then
23             AddResult($O$)
```

To address this issue, we need to extract only the relevant parts of the specific exercise when performing equivalence checking. Simply extracting a particular exercise based on the function name is not sufficient, as students may add their own helper functions or other definitions like addVote function in Listing 3.9. We use a simple fixed-point algorithm to extract specific exercise, as shown in Algorithm 1. At the start of the algorithm, both the worklist and result sets are initialized with the specified target definitions, which typically refer to functions pre-defined in the assignment framework that students need to complete. The algorithm then iterates through the definitions in the worklist for processing. If a definition is a class or trait, its parent and child classes are added to the workList for further processing. If the definition is a companion object, the corresponding class is added to the workList. Next, the algorithm extracts any other functions, classes, and companion objects that the current definition depends on, based on function calls, object construction, type constraints, and references to companion objects. New dependencies will be added to both the result and workList. The algorithm terminates once workList is empty.

Chapter 4

Performance Optimization

Stainless cannot guarantee a definitive result for every verification attempt. When verification conditions cannot be proven and no counterexample is found within the specified time, it returns a timeout as the result. In the context of equivalence checking, this means that the equivalence of the two programs is considered unknown. The reasons for a timeout include the use of undecidable theories, non-k-inductive invariants [17], and state explosion. In this chapter, we propose some optimization strategies for EPL programming exercises prone to state explosion to avoid unknown results by reducing the number of states.

4.1 Recursion Elimination

For recursive function calls or recursive ADT definitions, Stainless may repeatedly unfold their definitions until it can either prove them using k-induction or find a counterexample. As this process continues, the number of states in SMT formulas steadily increases [6], potentially causing the SMT solver to fail in returning a result due to the expanding search space, which can ultimately lead to a timeout. To mitigate the risk of timeouts, we will minimize recursive calls or avoid verifying recursive definitions whenever possible.

4.1.1 Substitute Map for ListMap

In Stainless, Map is directly supported by the Inox solver and utilizes decidable array theory [15]. During the verification process, Inox does not require any additional unfolding operations. In contrast, ListMap is defined as an ADT within the Stainless library. Both ListMap and the List implementation it relies on contain a substantial number of recursive definitions and recursive calls. This leads to a significant difference in verification efficiency between them.

```
/* Before */
import scala.collection.immutable.ListMap
def list2map[K,V](l: List[(K,V)]): ListMap[K,V] = l match {
  case Nil => ListMap[K,V]()
  case (k,v) :: m => list2map(m) + (k -> v)
}

/* After general translation */
import stainless.collection.{ListMap,List,Nil,Cons}
def list2map[K,V](l: List[(K,V)]): ListMap[K,V] = l match {
  // Nil and Cons objects in Stainless are slightly
  // different from those in Scala.
  case Nil() => ListMap[K,V]()
  case Cons((k,v),m) => list2map(m) + (k -> v)
}

/* After Using Map instead of ListMap for better performance */
import stainless.lang.Map
import stainless.collection.{List,Nil,Cons}
def list2map[K,V](l: List[(K,V)]): Map[K,V] = l match {
  case Nil() => Map[K,V]()
  case Cons((k,v),m) => list2map(m) + (k -> v)
}
```

Listing 4.1: An example of translating ListMap to Map in the list2map exercise.

Map in Stainless corresponds to scala.lang.Map, which is implemented by default as HashMap in Scala. The main semantic difference between HashMap and ListMap is that ListMap preserves the order of element insertion, whereas HashMap does not [18]. When the insertion order of elements is not important, the semantics of the two are effectively the same. Therefore, in the list2map and election exercises, where ListMap operations are challenging to verify, we translate Scala's ListMap into Stainless's Map as shown in Listing 4.1.

4.1.2 Avoid ADT Invariants in Recursive ADT

In Stainless, it is possible to add invariants to ADTs, and the system will automatically verify whether the invariant is violated at any occurrence of the ADT [14]. In Listing

3.2, the require statement we added in OverflowInt uses an ADT invariant to ensure that the underlying falls within the valid range for Int. However, we have observed that when an ADT with invariants, such as OverflowInt, appears within another recursively defined ADT, it may cause the recursive ADT's definition to be unfolded multiple times, leading to verification timeouts. Listing 4.2 shows a recursive ADT Expr used in the eval, tyOf, subst, and desugar exercises of Assignment 2. It represents the AST of a simple programming language called Giraffe. According to the general type conversion rules described in Section 3.1.1, the type of the n attribute in the Num subclass should be converted to OverflowInt at this point. However, this would be detrimental to verification in Stainless. A similar situation occurs with String. It is also converted into an ADT with invariants in Inox's internals.

```
/* Before */
type Variable = String
abstract class Expr
case class Num(n: Integer) extends Expr
case class Str(s: String) extends Expr
case class Plus(e1: Expr, e2: Expr) extends Expr
case class LetPair(x: Variable,y: Variable, e1:Expr, e2:Expr)
   extends Expr
...

/* After */
type Variable = BigInt
abstract class Expr
case class Num(n: BigInt) extends Expr
case class Str(s: BigInt) extends Expr
case class Plus(e1: Expr, e2: Expr) extends Expr
case class LetPair(x: Variable,y: Variable, e1:Expr, e2:Expr)
   extends Expr
...

/* Only using as symbolic value */
case Str(s) => StringV(s)
```

Listing 4.2: A specific translation for String and Integer in the framework code, applied to the eval, tyOf, subst, and desugar exercises.

In the eval, tyOf, subst, and desugar exercises of Assignment 2, students are not required to handle the specific uses of Int and String attributes within the Expr type. Instead, they are to treat these attributes as symbolic values. Consequently, we can translate them into the BigInt type which does not include additional invariants, as shown in Listing 4.2.

4.1.3 Recursive Call Elimination

The eval, tyOf, subst, and desugar exercises in EPL Programming Assignment 2 involve numerous self-recursive calls. Listing 4.3 provides an example of two equivalent eval functions, where a single case branch handling Apply in the first implementation includes three recursive calls. The correct eval function consists of 19 case branches for different subtypes of Expr, most of which contain multiple recursive calls. The other three exercises follow a similar pattern and include additional case branches. During equivalence checking, each recursive call might be unfolded by Stainless multiple times, leading to an explosion in the number of states.

```scala
/* Before */
def eval(env: Env[Type], e: Expr): Value = e match {
    ...
    case Apply(e1,e2) => eval(env,e1) match {
        ...
        case RecV(env0,f,x,e) =>
            eval(env0
            + (f->eval(env,e1))
            + (x->eval(env,e2)), e)
    }
}

/* Before */
def eval(env: Env[Type], e: Expr): Value = e match {
    ...
    case Apply(e1,e2) => eval(env,e1) match {
        ...
        case RecV(recEnv,f,x,recBody) =>
          eval(recEnv
            + (f -> RecV(recEnv,f,x,recBody))
            + (x -> eval(env,e2)), recBody)
      }
}
```

Listing 4.3: Two equivalent eval exercises.

```
@extern @pure
fake_eval(env: Env[Type], e: Expr): Value = errorWrapper

/* After */
def eval(env: Env[Type], e: Expr): Value = e match {
    ...
    case Apply(e1,e2) => fake_eval(env,e1) match
        case RecV(env0,f,x,e) =>
            fake_eval(env0
            + (f->fake_eval(env,e1))
            + (x->fake_eval(env,e2)), e)
        ...
}

/* After */
def eval(env: Env[Type], e: Expr): Value = e match {
    ...
    case Apply(e1,e2) => fake_eval(env,e1) match
        case RecV(recEnv,f,x,recBody) =>
          fake_eval(recEnv
            + (f -> RecV(recEnv,f,x,recBody))
            + (x -> fake_eval(env,e2)), recBody)
        ...
}
```

Listing 4.4: After eliminating recursive calls in the eval exercise from Listing 4.3

Fortunately, the patterns of recursive function calls in these exercises are fairly consistent. Two equivalent submissions typically indicate that they contain recursive calls with equivalent arguments. In Listing 4.3, although the two equivalent eval implementations have different numbers of recursive calls, there is a mapping relationship between them. The first, second, and fourth recursive calls in the first implementation correspond to the first, second, and third recursive calls in the second implementation. Additionally, the third recursive call also corresponds to the second recursive call in the second implementation, which makes them equivalent. Therefore, we can treat all recursive calls as invocations of the same external pure function as shown in Listing 4.4. Stainless can determine that the calls to fake_eval in the two implementations have a corresponding relationship by analyzing the differences in the parameters passed, thereby establishing their equivalence.

However, this method may yield incorrect non-equivalence results in certain cases.

Listing 4.5 shows an example. When retaining recursive calls, since both implementations handle Lambda by simply invoking desugar again on its sub-expressions, the first recursive calls in the LetFun branches are equivalent. However, after eliminating the recursive call, Stainless will no longer be able to capture the semantics of lambda handling, leading to the two implementations being considered non-equivalent. However, this method does not compromise the soundness of Stainless's equivalence checking, as it will not result in non-equivalent programs being considered equivalent.

```
/* Before */
def desugar(e: Expr): Expr = e match {
    ...
    case Lambda(x,ty,e) => Lambda(x,ty,desugar(e))
    case LetFun(f,arg,ty,e1,e2) =>
      Let(f,Lambda(arg,ty,desugar(e1)),desugar(e2))
}
/* After */
def desugar(e: Expr): Expr = e match {
    ...
    case LetFun(f,arg,ty,e1,e2) =>
      Let(f,
      Lambda(arg,ty,fake_desugar(e1)),
      fake_desugar(e2))
}

/* Before */
def desugar(e: Expr): Expr = e match {
    ...
    case Lambda(x,ty,e) => Lambda(x,ty,desugar(e))
    case LetFun(f,arg,ty,e1,e2) =>
      Let(f,desugar(Lambda(arg,ty,e1)),desugar(e2))
}
/* After */
def desugar(e: Expr): Expr = e match {
    ...
    case LetFun(f,arg,ty,e1,e2) =>
      Let(f,fake_desugar(Lambda(arg,ty,e1)), fake_desugar(e2))
}
```

Listing 4.5: An example of incorrect non-equivalence results after eliminating recursive calls.

4.2 Separate Verification

Aside from recursion, excessive pattern-matching case branches in the eval, tyOf, subst, and desugar exercises for handling all sub-types of Expr may also lead to a state explosion problem.

Algorithm 2: Subfunction Generation from Pattern Match

```
Input: A function containing a single pattern match (matchFun)
Output: Generated subfunctions (subFunctions)
1 $subFunctions \leftarrow \emptyset$
2 Assert matchFun is defined as:
3   def $F(a_1, \dots, a_n) = \{$
      $a_n$ match
        case $pat_1 \Rightarrow body_1$
        $\dots$
        case $pat_n \Rightarrow body_n$
    $\}$
4 foreach case $pat_i \Rightarrow body_i$ do
5   $newFun_i \leftarrow$ def $F_i(a_1, \dots, a_n) = \{$
      $a_n$ match
        case $pat_i \Rightarrow$ FixRecursiveCall($body_i$)
        case _ $\Rightarrow$ ErrorWrapper
    $\}$
6   $subFunctions \leftarrow subFunctions \cup newFun_i$
```

To reduce the complexity of individual functions in these exercises, we attempted to simplify these exercises by leveraging the fixed code patterns within them. Given that these exercises all include an outer pattern matching structure that handles different subtypes of Expr (as shown in Listing 4.3 and Listing 4.5), we can treat each case branch as an independent subfunction and convert the original equivalence check into an equivalence check between the corresponding subfunctions. The specific algorithm for generating sub-functions is shown in Algorithm 2. For eligible original functions, we will iterate through all their case branches and sequentially generate sub-functions that have the same signature but contain only a single Expr subtype case branch and a default case branch. The FixRecursiveCall method eliminates recursive function calls as described in Section 4.1.3. If this optimization is not enabled, it instead converts them into recursive calls to the sub-function itself. Listing 4.6 presents the sub-function generated from the Apply case branch of the first eval implementation in Listing 4.3.

```
/* After */
def eval_Apply(env: Env[Type], e: Expr): Value = e match {
    case Apply(e1,e2) => eval_Apply(env,e1) match
        case RecV(env0,f,x,e) =>
            eval_Apply(env0
            + (f->eval_Apply(env,e1))
            + (x->eval_Apply(env,e2)), e)
    case _ => errorWrapper
}
```

Listing 4.6: The subfunction corresponding first eval implementation in Listing 4.3.

At this point, the original function is considered equivalent only if all sub-functions that handle the same Expr subtype are equivalent. When non-corresponding sub-functions exist, the sub-function generated from the default case of the original function will be used for comparison. The approach to handling non-exhaustive matches, as described in Section 3.1.3, ensures that a default case is always present.

Similar to the recursive call elimination described in Section 4.1.3, separate verification can also produce incorrect non-equivalence results in the scenario shown in Listing 4.5 but does not compromise the soundness of Stainless's equivalence checking. This is also due to the loss of semantics from other case branches.

While separation verification is designed to address performance issues caused by state explosion, it also provides additional benefits. First, Stainless relies on counterexample search to prove the inequivalence between programs, and this process terminates as soon as a counterexample is found. However, for complex exercises of assignment 2, two assignments might be inequivalent in multiple case branches. A single counterexample is insufficient to comprehensively help human graders identify all the differing locations between them. With separate verification, however, we can obtain counterexamples for each corresponding inequivalent case branch. Furthermore, separate verification enables us to conduct clustering on the subfunctions generated by individual case branches instead of original functions. We will demonstrate the usefulness of this approach by evaluation in Section 7.4.

Chapter 5

Clustering Algorithm

The original Stainless clustering algorithm classifies student-submitted assignments solely as either equivalent to the reference answers or not [8]. While it supports setting multiple reference answers and can automatically expand the reference answer set using assignments equivalent to the reference answers during the clustering process, it is unable to cluster incorrect assignments. This will have certain limitations in practice. Human graders may not be able to provide enough reference answers to cover all correct assignments, which means that some correct answers might be incorrectly classified, necessitating additional manual review to identify them. Furthermore, human graders often need to recognize which incorrect answers are equivalent in order to provide consistent scoring and feedback. Listing 5.1 shows an example. The last two update implementations are not equivalent to the first reference solution because they throw an error when given an empty list as input. However, they can still be considered correct, as the assignment instructions do not explicitly state how students should handle an empty list. It would be unreasonable to classify these two solutions as incorrect. Furthermore, in the original stainless clustering, these two updates are grouped with other entirely incorrect implementations, forcing human graders to expend additional effort to manually identify them.

To address this, we extended the original Stainless clustering algorithm, as shown in Algorithm 3. The StainlessCluster function represents the original Stainless clustering algorithm. It takes a set of reference models and candidates as input and returns three clusters: one containing candidates that are equivalent to the reference models, another with candidates that cannot be safely checked for equivalence, and a third with candidates whose function signatures differ from those of the reference models. In the first round of clustering, the reference answers serve as the models. In each subsequent round, the first candidate from the remaining set is used as the initial reference model, and this process repeats until the candidate set is empty. The original Stainless clustering algorithm is sound, so after each round, we can remove the elements in the cluster from the candidate set, as they cannot be equivalent to any other candidates.

```
// Reference solution
def update[K, V](m: List[(K, V)], k: K, v: V): List[(K, V)] =
    m match
        case Nil => List((k,v))
        case (k2,v2)::m2 => if(k == k2) (k,v)::m2 else (k2,v2)::
            update(m2,k,v)

// Not equivalent to the reference answer, but can be considered
    correct.
def update[K, V](m: List[(K, V)], k: K, v: V): List[(K, V)] =
    m match
        case Nil => sys.error("key not in list")
        case x :: xs => if (x._1 == k) {(k,v) :: xs} else {x ::
            update(xs,k,v)}

// Equivalent to the previous update.
def update[K, V](m: List[(K, V)], k: K, v: V): List[(K, V)] =
    m match
        case Nil => sys.error("Key not found in map")
        case (key,value) :: xs => if (key == k) {(k,v) :: xs} else
            {(key,value) :: update(xs,k,v)}
```

Listing 5.1: Two correct assignments that are not equivalent to the sample solution.

Algorithm 3: Clustering Based on Equivalence with Reference Model

Input : Reference answers $R_1,\dots,R_n$ and student submissions $S_1,\dots,S_n$

Output : Clusters of equivalent submissions (*clusters*)

```
1 $candidates \leftarrow \{S_1,\dots,S_n\}$
2 $models \leftarrow \{\mathrm{R}_1,\dots,R_n\}$
3 $clusters \leftarrow \emptyset$
4 while $candidates \neq \emptyset$ do
5     $cluster, unsafe, wrong \leftarrow$ StainlessCluster($models, candidates$)
6     $candidates \leftarrow candidates \setminus cluster \setminus unsafe \setminus wrong$
7     $clusters \leftarrow clusters \cup \{cluster\}$
8     $models \leftarrow$ {pop one element from $candidates$}
```

Chapter 6

Implementation

6.1 System Overview

Our implementation is a further development based on Stainless and Scala compiler it depends, which is open-source and available at `https://github.com/LioTree/stainless-epl`. The overall architecture is illustrated in Figure 6.1. The red dashed boxes represent the newly introduced components, cluster and transformer, which are detailed in Section 6.2 and Section 6.3 respectively. The blue dashed boxes indicate the original Stainless components that we extended and modified, as discussed in Section 6.4.

During system execution, the cluster module, which implements the clustering algorithm from Chapter 5, will invoke the Scala compiler and pass in the Scala source code from both the students' assignments and the reference solutions. It also uses command-line arguments to inform the Target Extractor about which exercise to extract and to instruct the equivchk module in Stainless on which assignments need to be clustered in the current round. Both the Stainless and Scala libraries are compiled alongside input source code. The transformer operates on the untyped AST produced by the parser, performing the Scala to Stainless Scala translation and additional transformations as described in Chapter 3 and Chapter 4. Following the pickler phase of the compiler, the Scala AST will be transformed into the AST of Stainless's internally supported purely functional verification language. After this, Stainless's Equivchk component generates verification conditions for equivalence checking and passes them to the Inox solver for verification. The cluster script will decide whether to proceed to the next round of clustering and which assignments will participate in the next round based on the output of Stainless.

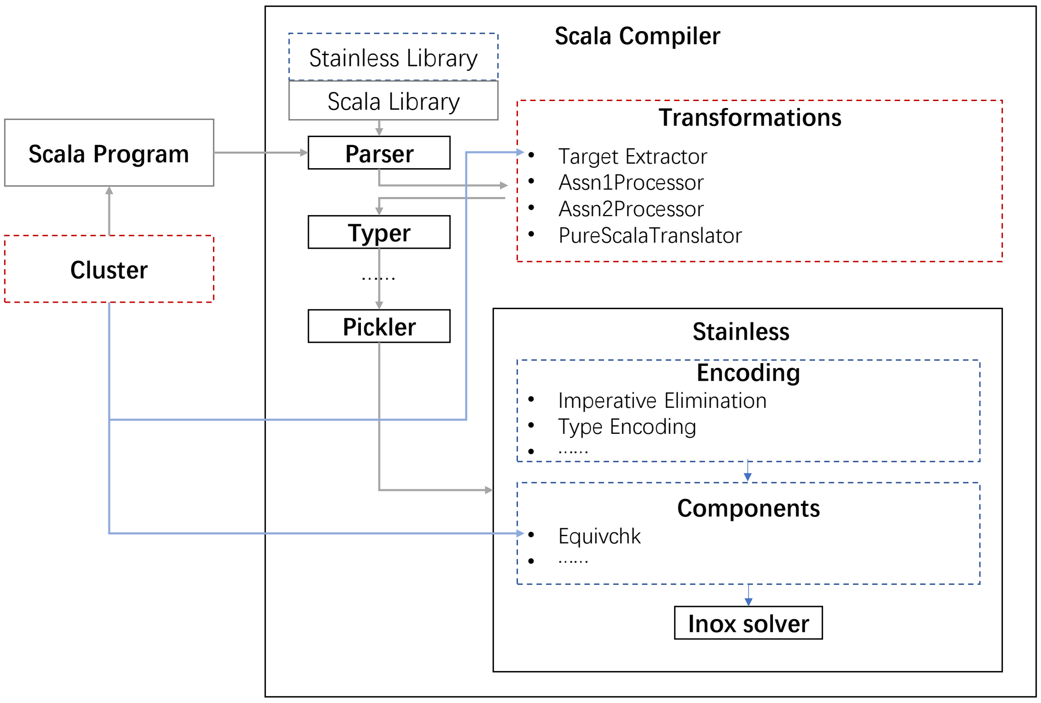

Figure 6.1: System Overview

6.2 Cluster

We implemented the clustering algorithm described in Chapter 5 using an external cluster Python script. This algorithm relies on Stainless's existing clustering algorithm, so it was unnecessary to modify Stainless internally. Implementing it with an external Python script was more straightforward. Although this approach requires recompiling and transforming the assignment source code for each clustering round, the overhead is acceptable given the relatively small code size of the programming assignments we are processing.

6.3 Transformers for Untyped AST

The transformer includes the Target Extractor, PureScalaTranslator, Assn1Processor and Assn2Processor. They will perform transformations on the untyped AST. The Target Extractor implements the extraction algorithm described in Section 3.2, while the PureScalaTranslator implements the translation rules described in Section 3.1.1 and Section 3.1.3. Their uses are not limited to equivalence checking and clustering of programming assignments. The former can also be used to extract and verify specific portions of relevant code, while the latter can simply be used to convert standard Scala into the subset of pure Scala supported by Stainless. The transformations specific to certain exercises, as described in Section 3.1.2 and Chapter 4, are handled by Assn1Processor and Assn2Processor.

The choice to use an untyped AST was primarily motivated by the need for development efficiency. Typed AST in the Scala compiler are considerably more complex, and converting them demands significantly more development effort. Furthermore, the format of the EPL programming assignments we deal with is relatively fixed, meaning the imprecision of untyped AST rarely leads to any negative impact.

Moreover, employing an untyped AST for the target extractor provides the added benefit of managing cases where students' submissions might include code that fails type checking. The untyped AST-based target extractor can still discard these exercises, except in instances where they contain syntax errors that the parser cannot process.

6.4 Extension of Stainless

We extended the Stainless library to better support the language constructs used in EPL programming assignments. This effort included adding the OverflowInt class described in Section 3.1.1 and the errorWrapper function outlined in Section 3.1.3. Additionally, we introduced a StringWrapper class to replace String, which provides some common operations missing from Stainless String, such as concatenation with integers. We also added some missing methods to the Stainless ListMap and List classes to align them more closely with the Scala library. These extensions to the library are also applicable to other uses of Stainless, such as writing formally verified critical components directly in Stainless Scala.

To support the conversion of original function equivalence into the equivalence of multiple subfunctions, as proposed in Section 4.2, we added a new subfunction equivalence strategy to Stainless's existing Equivchk component. When this strategy is enabled, the Equivchk component will automatically identify and match corresponding subfunctions of original functions, determining the equivalence of the original function based on the equivalence of these subfunctions.

Additionally, Stainless's type encoding replaces function calls that return Nothing with calls to stainless.lang.error[Nothing], which causes calls to errorWrapper in SMT expressions to still be treated as calls to different uninterpreted functions. We made slight modifications to handle calls to errorWrapper specifically.

Chapter 7

Evaluation

In this chapter, the overall system is evaluated using real EPL programming assignments. We will introduce the dataset and testing environment used for our evaluation in Section 7.1. Sections 7.2, 7.3, and 7.4 will respectively discuss the coverage of supported EPL programming assignments, runtime performance and the impact of performance optimization strategies, as well as the accuracy and usefulness of the clustering results.

7.1 Benchmarks Setup

We evaluated using historical assignments submitted by students in the 2023 EPL course. Assignment 1 comprised a total of 38 submissions and 1 reference sample solution, while Assignment 2 had 30 submissions and 1 reference sample solution. We performed clustering on the exercises in these two assignments separately. The tests were conducted on a system with a 12th Gen Intel Core i9-12900H CPU (20 cores, 2 threads per core), 16 GB of RAM, running Debian OS in 64-bit mode. We used the default Z3 SMT solver in Stainless, setting its timeout to 2 seconds and the overall timeout to half an hour. The reason for setting an overall timeout duration is that we discovered Stainless may hang and fail to terminate during the verification of equivalence between two programs after multiple timeouts, even if the SMT solver's timeout is set. The final evaluation results are presented in Table 7.1 and Table 7.2.

7.2 Coverage of Supported Assignments

Our initial focus is on the coverage of supported assignments within each exercise. This depends on the effectiveness of the transformation rules and target extraction algorithms we discussed in Chapter 3, as well as how well they are implemented and how effectively the stainless library is extended. We recorded the number of valid assignments for each exercise and the number that Stainless could process after transformation. Valid here refers to assignments that do not contain compilation errors. Assignments that could not be clustered due to unprovable termination or incorrect function signatures are also included because we are not focusing on the clustering results here. The percentage of supported assignments among the valid ones reflects the coverage rate of supported assignments (SA/VA columns in Table 7.1 and Table 7.2). In Assignment 1, all exercises except for the election exercise (79%) achieved a coverage rate of over 90%. In Assignment 2, all exercises reached a coverage rate of over 80%. The average SA/VA after applying performance optimization strategies across all exercises was 96%. The main reason for the lack of support is that certain language features are not supported by Stainless and cannot be addressed through transformations or extensions to the Stainless library. Typical examples include lambda functions with effects and the use of floating-point numbers.

Additionally, the implementation of the transformer for untyped ASTs resulted in a few extra errors. In the filter exercise of Assignment 1, an error occurred because our target extractor could not distinguish between local variables and external definitions with the same name, leading to the extraction of unrelated exercises that used unsupported features. In Assignment 2, because our target extractor can only extract top-level definitions, the first four exercises within the same top-level definition had to be extracted together. As a result, unsupported features present in Value.length and Value.index in four submissions affected the other two exercises.

To further explore the significance of the target extractor, we conducted an ablation study by disabling it. In this case, only assignments where all exercises are valid and can be supported are processed. Under these conditions, even without using the performance optimization strategy which reduces the number of supported assignments, only 22 assignments in Assignment 1 and 18 assignments in Assignment 2 could be processed. This is significantly lower than the average number of supported assignments when the target extractor is enabled, which are 36 and 26, respectively.

Name	VA	SA	SA/VA	CA	CL	TC	TOR	CL-TC	Time
polynomial	39	38	97%	38	5	5	0	0	1m18s
sum	39	36	92%	29	1	1	0	0	44s
cycle	38	38	100%	34	2	2	0	0	14s
suffix	39	39	100%	38	6	6	0	0	1m18s
boundingBox	38	38	100%	37	4	4	0	0	44s
mayOverlap	36	36	100%	36	18	18	0	0	5m1s
compose1	36	36	100%	36	2	2	0	0	14s
compose	38	37	97%	37	2	2	0	0	29s
map	39	38	97%	38	2	2	0	0	17s
filter	37	35	95%	35	3	3	0	0	38s
reverse	39	38	97%	38	7	2	5	5	21m15s
lookup	39	39	100%	37	2	2	0	0	41s
update	38	36	95%	35	11	9	4	2	5m41s
keys	39	39	100%	38	3	2	1	1	1m36s
presidentListMap	37	37	100%	36	4	4	0	0	1m18s
map12	37	36	97%	36	5	5	2	0	2m42s
list2map	39	39	100%	37	5	3	5	2	18m46s
list2map$_{map}$	39	36	92%	34	5	3	2	2	1m49s
election	39	36	92%	34	7	3	6	4	40m40s
election$_{map}$	39	31	79%	31	7	3	4	4	14m25s

Table 7.1: The evaluation result for EPL programming assignment 1. The VA column displays the number of valid assignments for this exercise, while the SA column indicates the number of assignments accepted by Stainless after applying the transformation and utilizing the extended Stainless library. The SA/VA ratio reflects the effectiveness of these processes. CA represents the final number of assignments that participate in the clustering, excluding those with incorrect function signatures and those that cannot be proven to terminate by Stainless. The following four rows pertain to the clustering outcomes: the CL column denotes the number of clusters we identified, and TC represents the true number of clusters after manual verification. The TOR and CL-TC columns detail the number of rounds that resulted in an unknown outcome due to a Z3 timeout (TOR) and the difference between the number of clusters obtained and the true number of clusters. The clustering results are accurate when both of them are 0. The final Time column represents the total time of clustering. In the Name column, the map subscript denotes the application of a performance optimization strategy that replaces ListMap with Map.

Name	VA	SA	SA/VA	CA	CL	TC	TOR	CL-TC	Time
Value.multiply					1	1	0	0	24s
Value.eq					3	3	1	2	1m18s
Value.length	30	26	87%	26	1	1	0	0	29s
Value.index					-	-	-	-	Timeout
Value.concat					1	1	0	0	25s
eval					-	-	-	-	Timeout
$\text{eval}_{re,sv}$					-	-	-	-	Timeout
$\text{eval}_{ai,sv}$	29	29	100%	27	-	-	-	-	Timeout
$\text{eval}_{re,ai}$					16	16	0	0	11m40s
$\text{eval}_{re,ai,sv}$					16	16	0	0	12m10s
tyOf					-	-	-	-	Timeout
$\text{tyOf}_{re,sv}$					-	-	-	-	Timeout
$\text{tyOf}_{ai,sv}$	28	23	82%	23	-	-	-	-	Timeout
$\text{tyOf}_{re,ai}$					21	21	9	0	21m8s
$\text{tyOf}_{re,ai,sv}$					21	21	0	0	14m3s
subst					-	-	-	-	Timeout
$\text{subst}_{re,sv}$					-	-	-	-	Timeout
$\text{subst}_{ai,sv}$	29	29	100%	29	-	-	-	-	Timeout
$\text{subst}_{re,ai}$					24	24	0	0	9m38s
$\text{subst}_{re,ai,sv}$					24	24	0	0	17m52s
desugar					-	-	-	-	Timeout
$\text{desugar}_{re,sv}$					-	-	-	-	Timeout
$\text{desugar}_{ai,sv}$	29	28	97%	28	-	-	-	-	Timeout
$\text{desugar}_{re,ai}$					20	17	0	3	6m39s
$\text{desugar}_{re,ai,sv}$					20	17	0	3	10m48s

Table 7.2: The evaluation result for EPL Programming Assignment 2. The columns have the same meaning as in Table 7.1. The subscripts re, ai, and sv represent enabling recursive call elimination, avoiding ADT invariants, and the separate verification strategy, respectively. The eval, tyOf, subst, and desugar exercises using different optimization strategies yielded the same results in VA, SA, SA/VA, and CA. Timeout refers to cases where the total duration exceeded the overall time limit we set in Section 7.1.

7.3 Runtime Performance

We assessed the performance of our system by measuring the total time taken for the transformation and clustering of each exercise. We did not record the time for each process separately, as clustering is much more complex than transformation, and the time required for the latter is almost negligible. The results are shown in the Time column of Table 7.1 and Table 7.2. After implementing the optimization strategy of using Map instead of ListMap, the total clustering time for the list2map and election exercises in Assignment 1 significantly decreased from 18m46s to 1m49s and from 40m40s to 14m25s, respectively. While both exercises produced the same number of clusters and the number of assignments that could be processed slightly decreased with this strategy, the rounds of clustering that contained unknown results were reduced by 3 rounds for list2map and by 2 rounds for election, which means obtaining more counterexamples that demonstrate the non-equivalence of different clusters. The Value.index exercise in Assignment 2 encountered timeouts in total time, with no applicable performance optimization strategies available for them. This might result from string operations that are difficult to verify. The eval, tyOf, subst, and desugar exercises were able to resolve timeout issues by simultaneously applying the strategies of avoiding ADT invariants, recursion elimination, and separate verification. To assess the impact of each optimization strategy, we also tested the outcomes when each strategy was disabled individually for these four exercises. The results indicated that in all four exercises, disabling either the avoidance of ADT invariants or recursion elimination led to timeouts, even with other optimization strategies still in place. Disabling separate verification resulted in more clustering rounds containing unknown results in the tyOf exercise, likely due to the need for students to perform sub-pattern matching in each case branch, which made the number of states large in the original function.

7.4 Accuracy and Usefulness of Clustering

Stainless's equivalence checking combines inductive proofs with counterexample search, offering a formally guaranteed deterministic outcome for the transformed program. Even in the presence of unknown results, the equivalence checking and clustering remain sound, as they do not cause inequivalent assignments to be mistakenly classified as equivalent. However, if too much of the original semantics is lost during the transformation process, the clustering of the original program may no longer possess these two properties. To evaluate this, we assessed the accuracy of the clustering by calculating the difference between the number of clusters obtained and the actual number of clusters (CL-TC in Table 7.1 and Table 7.2) for each exercise and examined the cases where the two were not equivalent. The actual number of clusters was determined by referencing real EPL assignment grades and additional manual review. The result shows that all the clustering results were sound, with no cases of inequivalent assignments being placed in the same cluster. Additionally, all incomplete results, except for the desugar exercise using recursive call elimination and separate verification strategies, were caused by unknown outcomes in the equivalence checking. The incomplete results observed in desugar are also not surprising. Their occurrence is consistent with the shortcomings of the two strategies we described in Section 4.1.3 and Section 4.2.

In addition to accuracy, the number of clusters is also relevant to the usefulness of the clustering algorithm. As indicated in the CL column of Table 7.2, in all exercises of Assignment 1 and the first four exercises of Assignment 2, the number of clusters is relatively small. Even in the exercise with the most clusters, mayOverlap, each cluster contains an average of only two assignments. However, we observed a relatively high number of clusters in the eval, tyOf, subst, and desugar exercises of Assignment 2, with an average of less than two assignments per cluster. Although these clustering results are semantically correct, except for three additional clusters in desugar, the excessive number of clusters makes them impractical for human graders since they would still need to spend a significant amount of time reviewing each assignment individually. Our separate verification strategy helps in this regard. We employed a separate verification approach to individually cluster the subfunctions generated from these four exercises. The results, presented in Table 7.3, show that most subfunctions from these exercises form relatively few clusters. This enables human graders to efficiently complete their grading and provide feedback based on the clustering outcomes. A few subfunctions, such as those handling Rec and LetRec in the tyOf exercise, LetRec in the subst exercise, and LetPair in the desugar exercise, have more than ten clusters. However, the maximum number of clusters, which is 16, is still comparable to the smallest number of clusters in the eval exercise (as shown in Table 7.2), and the complexity of these subfunctions remains significantly lower than that of the original functions since they only handle one type. This approach of clustering separately is also more closely aligned with the strategy typically employed by human graders when evaluating such assignments: scoring each branch individually and providing feedback, then calculating the overall total score, rather than simply giving an aggregate score for the whole exercise.

Name	eval			tyOf			subst			desugar		
	CL	TC	CL-TC	CL	TC	CL-TC	CL	TC	CL-TC	CL	TC	CL-TC
Num	1	1	0	1	1	0	1	1	0	1	1	0
Plus	1	1	0	1	1	0	1	1	0	1	1	0
Minus	1	1	0	1	1	0	1	1	0	1	1	0
Times	1	1	0	2	2	0	1	1	0	1	1	0
Bool	1	1	0	2	2	0	2	2	0	2	2	0
Eq	1	1	0	4	4	0	3	3	0	2	2	0
IfThenElse	4	4	0	6	6	0	3	3	0	2	2	0
Str	2	2	0	2	2	0	2	2	0	2	2	0
Length	1	1	0	2	2	0	2	2	0	2	2	0
Index	1	1	0	3	3	0	2	2	0	2	2	0
Concat	1	1	0	2	2	0	2	2	0	2	2	0
Var	3	3	0	2	2	0	1	1	0	2	2	0
Let	2	2	0	1	1	0	1	1	0	2	2	0
Pair	2	2	0	2	2	0	2	2	0	2	2	0
First	3	3	0	4	4	0	4	4	0	3	3	0
Second	3	3	0	4	4	0	4	4	0	3	3	0
Lambda	2	2	0	4	4	0	4	4	0	2	2	0
Rec	4	4	0	15	15	0	9	9	0	2	2	0
Apply	9	9	0	4	4	0	2	2	0	2	2	0
LetPair	-	-	-	2	2	0	8	8	0	16	14	2
LetFun	-	-	-	3	3	0	9	9	0	5	3	2
LetRec	-	-	-	11	11	0	14	14	0	6	5	1

Table 7.3: The results of clustering the sub-functions generated in the eval, tyOf, subst, and desugar exercises of Assignment 2. CL and TC have the same meanings as in 7.1.

Chapter 8

Related Work

8.1 Assignment Grading Based on Stainless

Milovancevic, Bucev, Wojnarowski, *et al.* [9] also applied Stainless's equivalence checking for the automated grading of real Scala programming assignments. The programming assignments they focused on are comparable in difficulty to EPL Programming Assignment 1, which includes tasks like removing the nth element from a list. However, their assignments consist of only four exercises. In contrast, the EPL assignments we need to tackle, particularly Assignment 2, are significantly more complex. Unlike our approach, which involves transforming standard Scala code, they required students to complete their assignments directly using the Stainless library. While this ensures that the submitted assignments are compatible with Stainless, it may pose additional challenges for students who wish to run and debug their code locally, as they would need to configure the Stainless environment and include the library during compilation. Similar to our approach, they also clustered student submissions to make it easier for manual graders to provide scores and feedback to entire clusters. However, their clustering method involves running equivalence checks on each pair of submissions sequentially. This approach fails to leverage the transitivity offered by Stainless's equivalence checking, leading to many redundant comparisons.

8.2 Program Equivalence and Clustering in Education

CLARA [3] uses dynamic equivalence to cluster programming assignments written in Python and Java. It executes programs dynamically based on the provided test cases and checks for a one-to-one correspondence between variables and control flow structures.

It is primarily focused on imperative programs, considering only loops and not recursion. Although Olofsson [19] extended Clara to support recursive functions, their approach still has limitations in our context. Firstly, their approach still relies on complete test cases. In EPL programming assignment 2 we are considering, providing comprehensive test cases is challenging because the inputs are Giraffa programs that must adhere to specific grammar rules. Secondly, their results lack formal guarantees. Potential errors in clustering could force human graders to invest considerable effort in manually reviewing the outcomes to avoid incorrect grading and feedback. Additionally, they adopt strict structural equivalence, which may result in an excessive number of clusters for more advanced assignments like the EPL programming assignment. SemCluster [4] broadens the definition of structural equivalence by collecting data flow and control flow features during dynamic execution and clustering them using the k-means algorithm. This allows structurally similar but not identical programs to be grouped. However, their approach still faces two other issues.

TESTML [5] employs bounded symbolic execution to assess whether OCaml assignments are equivalent to a reference solution. It is similar to our approach in its support for functional languages, as it can handle features such as recursion, higher-order functions, and lambda functions. However, due to the limitations of bounded symbolic execution techniques, it can only find counterexamples within a limited scope and lacks the capability to prove equivalence. As a result, it is unable to effectively cluster equivalent assignments and may produce unsound results.

ZEUS [7] provides clustering with formal guarantees for purely functional Standard ML assignments. Similar to Clara, it also requires structural equivalence. Unlike the function induction method used by Stainless, ZEUS uses inference rules between the expressions representing two programs. This imposes some limitations on its support for recursion. Additionally, it does not provide counterexamples.

Vujosevic-Janicic and Maric [6] determine the equivalence of C/C++ programming assignments based on verification tool LAV [20]. Their verification approach is similar to the Stainless-based method we use, as both include counterexample findings and proofs. Although it supports recursion, it is primarily designed for imperative languages and therefore lacks support for features such as higher-order functions. Moreover, it does not implement assignment clustering.

Chapter 9

Conclusions

In this dissertation, we propose and implement a framework for clustering EPL programming assignments 1 and 2 using equivalence checking based on the Stainless formal verification tool. This is accomplished by transforming Scala's AST to convert standard Scala into Stainless Scala and by applying performance optimization strategies to certain exercises to reduce the difficulty of verification. Additionally, we introduce a clustering algorithm that clusters student submissions with equivalent semantics, enabling human graders to provide grading and feedback for multiple assignments simultaneously. By evaluating our approach on real EPL programming assignments, we have validated the effectiveness of our solution.

One of the main limitations of our work is that the entire framework is specifically designed for EPL programming assignments 1 and 2, so it may not be directly applicable to other Scala programming assignments. Stainless is not intended to achieve fully automated proofs. For complex verification, it still relies on user-provided hints and code modifications [14]. Therefore, the manual development of optimized strategies tailored to specific complex assignments is necessary to avoid excessive timeouts. Fully automating this process may prove to be quite challenging. However, our framework can be extended to support other Scala programming assignments by developing new transformers specifically designed for them. This approach is particularly well-suited for assignments like EPL programming assignments 1 and 2, which remain consistent year after year.

Bibliography

[1] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *Journal on Educational Resources in Computing*, vol. 5, no. 3, p. 4, Sep. 2005, ISSN: 1531-4278, 1531-4278. DOI: 10.1145/1163405.1163409. [Online]. Available: https://dl.acm.org/doi/10.1145/1163405.1163409 (visited on 08/22/2024).

[2] M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak, "Software verification and graph similarity for automated evaluation of students' assignments," *Information and Software Technology*, vol. 55, no. 6, pp. 1004–1016, Jun. 2013, ISSN: 09505849. DOI: 10.1016/j.infsof.2012.12.005. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584912002406 (visited on 08/22/2024).

[3] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 465–480, Jun. 11, 2018, ISSN: 0362-1340. DOI: 10.1145/3296979.3192387. [Online]. Available: https://dl.acm.org/doi/10.1145/3296979.3192387 (visited on 04/17/2024).

[4] D. M. Perry, D. Kim, R. Samanta, and X. Zhang, "SemCluster: Clustering of imperative programming assignments based on quantitative semantic features," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, New York, NY, USA: Association for Computing Machinery, Jun. 8, 2019, pp. 860–873, ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314629. [Online]. Available: https://dl.acm.org/doi/10.1145/3314221.3314629 (visited on 03/17/2024).

[5] D. Song, M. Lee, and H. Oh, "Automatic and scalable detection of logical errors in functional programming assignments," *Proceedings of the ACM on Programming Languages*, vol. 3, 188:1–188:30, OOPSLA Oct. 10, 2019. DOI:

10.1145/3360614. [Online]. Available: https://dl.acm.org/doi/10.1145/3360614 (visited on 03/18/2024).

[6] M. Vujosevic-Janicic and F. Maric, "Regression verification for automated evaluation of students programs," *Computer Science and Information Systems*, vol. 17, no. 1, pp. 205–227, 2020, ISSN: 1820-0214, 2406-1018. DOI: 10.2298/CSIS181220019V. [Online]. Available: https://doiserbia.nb.rs/Article.aspx?ID=1820-02141900019V (visited on 04/06/2024).

[7] J. Clune, V. Ramamurthy, R. Martins, and U. A. Acar, "Program equivalence for assisted grading of functional programs," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–29, OOPSLA Nov. 13, 2020, ISSN: 2475-1421. DOI: 10.1145/3428239. [Online]. Available: https://dl.acm.org/doi/10.1145/3428239 (visited on 01/31/2024).

[8] D. Milovančević and V. Kunčak, "Proving and Disproving Equivalence of Functional Programming Assignments," *Proceedings of the ACM on Programming Languages*, vol. 7, 144:928–144:951, PLDI Jun. 6, 2023. DOI: 10.1145/3591258. [Online]. Available: https://dl.acm.org/doi/10.1145/3591258 (visited on 03/18/2024).

[9] D. Milovancevic, M. Bucev, M. Wojnarowski, S. Chassot, and V. Kuncak, "Formal Autograding in a Classroom," [Online]. Available: https://infoscience.epfl.ch/server/api/core/bitstreams/05b7cb84-c267-4895-a656-31ca62febf9a/content (visited on 08/21/2024).

[10] EPFL. "Epfl-lara/inox: Solver for higher-order functional programs." (), [Online]. Available: https://github.com/epfl-lara/inox (visited on 04/19/2024).

[11] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, red. by D. Hutchison, T. Kanade, J. Kittler, *et al.*, vol. 4963, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78799-0 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24. [Online]. Available: http://link.springer.com/10.1007/978-3-540-78800-3_24 (visited on 08/20/2024).

[12] H. Barbosa, C. Barrett, M. Brain, *et al.*, "Cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 13243, Cham: Springer International Publishing, 2022, pp. 415–

442, ISBN: 978-3-030-99523-2 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24. [Online]. Available: https://link.springer.com/10.1007/978-3-030-99524-9_24 (visited on 08/20/2024).

[13] V. Kunčak and J. Hamza, "Stainless Verification System Tutorial," Oct. 2021. DOI: 10.34727/2021/ISBN.978-3-85448-046-4_2. [Online]. Available: https://repositum.tuwien.at/handle/20.500.12708/18609 (visited on 03/24/2024).

[14] "Stainless documentation — Stainless 0.9.1 documentation." (), [Online]. Available: https://epfl-lara.github.io/stainless/index.html (visited on 08/21/2024).

[15] N. C. Y. Voirol, "Verified Functional Programming," DOI: 10.5075/epfl-thesis-9479. [Online]. Available: http://doi.org/10.5075/epfl-thesis-9479 (visited on 08/21/2024).

[16] R. W. Blanc, "Verification by Reduction to Functional Programs," DOI: 0.5075/epfl-thesis-9479. [Online]. Available: http://doi.org/10.5075/epfl-thesis-9479 (visited on 08/21/2024).

[17] S. Gambhir and V. Kunc ˇak, "CHC solvers in Stainless: Work in Progress," [Online]. Available: https://www.sci.unich.it/hcvs24/papers/HCVS2024_paper_9.pdf (visited on 08/21/2024).

[18] "Scala Standard Library 2.13.6 - scala.collection.immutable.ListMap." (), [Online]. Available: https://www.scala-lang.org/api/2.13.6/scala/collection/immutable/ListMap.html (visited on 08/23/2024).

[19] F. Olofsson, "Clustering and Matching Student Solutions in Source Academy," [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1787806/FULLTEXT01.pdf (visited on 08/21/2024).

[20] M. Vujošević-Janičić and V. Kuncak, "Development and Evaluation of LAV: An SMT-Based Error Finding Platform: System Description," in *Verified Software: Theories, Tools, Experiments*, vol. 7152, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 98–113, ISBN: 978-3-642-27704-7 978-3-642-27705-4. DOI: 10.1007/978-3-642-27705-4_9. [Online]. Available: http://link.springer.com/10.1007/978-3-642-27705-4_9 (visited on 08/23/2024).