Automated Equivalence Checking With Stainless

Dragana Milovančević, EPFL

December 8, 2022

Why Equivalence Checking?

- Because formal verification is hard!

```
def popMin(n: Node): (BigInt, Tree) = {
  require(isValidRBSubtree(n))
  decreases(n.size)
  n match
    case Node(R, Empty(B), v, r) => ...
    case Node(B, Empty(B), v, r) => ...
    case Node(c, l: Node, v, r) => ...
} ensuring(res => res._1 :: res._2.toList == n.toList
  && isValidTempSubtree(res._2)
  && (res._2.color == BB || redNodesHaveBlackChildren(res._2))
  && ((n.c == B) ==> (res._2.color != R))
  && blackHeight(res._2) == blackHeight(n)
)
```

- Applications in high-performance computing, software evolution, translation validation, automated grading...

Exercise

Define a function that implements fast exponentiation using a logarithmic number of steps.

Exercise

Define a function that implements fast exponentiation using a logarithmic number of steps.

$$b^{2n} = (b^2)^n = (b^n)^2$$

$$b^{2n+1} = b * b^{2n}$$

Exercise

Define a function that implements fast exponentiation using a logarithmic number of steps.

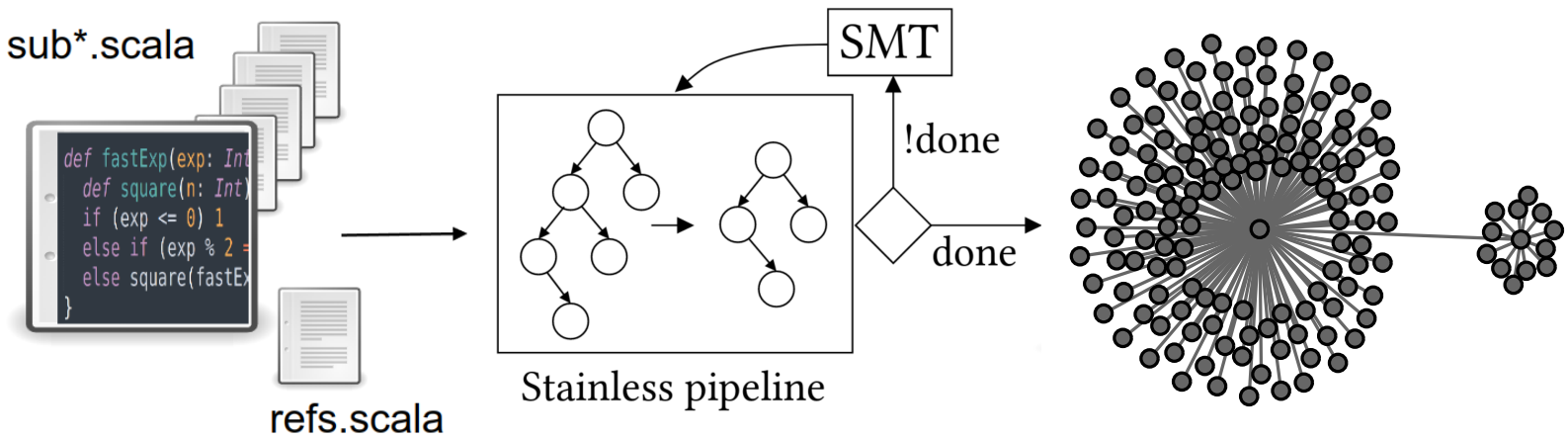
$$b^{2n} = (b^2)^n = (b^n)^2$$

$$b^{2n+1} = b * b^{2n}$$

```
def square(n: Int): Int = n*n
def fastExp(exp: Int, base: Int): Int =
  if exp <= 0 then 1
  else if exp % 2 == 0 then square(fastExp(exp / 2, base))
  else square(fastExp((exp - 1) / 2, base)) * base
```

Question: Is this solution correct?

Automated Equivalence Checking with Stainless



Example 0

- exercises/0_fastexp.scala

Example 0

- exercises/0_fastexp.scala
- Homework:
  Prove the equivalence of the four reference solutions

Overview of the Underlying Approach

Overview of the Underlying Approach

3. Clustering algorithm that finds the subset of correct solutions

2. Function call matching based on type- and test-directed search

1. Pairwise equivalence checking based on functional induction

Pairwise Equivalence Checking

Pairwise Equivalence Checking

A candidate program F is equivalent to a reference program M if:

- M and F have the same signature
- M and F terminate
- M and F return the same output for all inputs

Example 1

- exercises/1_sorted.scala
- Task:
  Help Stainless prove that isSortedB(l) == isSortedR(l)

Pairwise Equivalence Checking: An Example

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

def isSortedM(l: List[Int]): Boolean =
  l match
    case Nil() => true
    case Cons(x, Nil()) => true
    case Cons(x, Cons(y, xs)) if (x <= y) => isSortedM(y::xs)
    case _ => false
```

Pairwise Equivalence Checking: Functional Induction

```
// a copy of isSortedF
def indProof(l: List[Int]): Boolean = {
  if l.isEmpty then true
  else if !l.tail.isEmpty && l.head > l.tail.head then false
  else indProof(l.tail)
} ensuring(isSortedF(l) == isSortedM(l)) // result == isSortedM

def lemma(l: List[Int]): Unit = {
  val proof = indProof(l)
} ensuring(isSortedF(l) == isSortedM(l))
```

Pairwise Equivalence Checking: Functional Induction

```
// a copy of isSortedF
def indProof(l: List[Int]): Boolean = {
  if l.isEmpty then true
  else if !l.tail.isEmpty && l.head > l.tail.head then false
  else indProof(l.tail)
} ensuring(isSortedF(l) == isSortedM(l)) // result == isSortedM

def lemma(l: List[Int]): Unit = {
  val proof = indProof(l)
} ensuring(isSortedF(l) == isSortedM(l))
```

Pairwise Equivalence Checking: Unfolding and Functional Induction

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









} ensuring(isSortedF(l) == isSortedM(l))
```

Pairwise Equivalence Checking: Unfolding and Functional Induction

```
def indProof(l: List[Int]): Boolean = {
  if l.isEmpty then true
  else if !l.tail.isEmpty && l.head > l.tail.head then false
  else
    val t = l.tail
    if t.isEmpty then true
    else if !t.tail.isEmpty && t.head > t.tail.head then false
    else indProof(t.tail)
    assume(isSortedF(t) == isSortedM(t))
} ensuring(isSortedF(l) == isSortedM(l))
```

Pairwise Equivalence Checking: Unfolding and Functional Induction

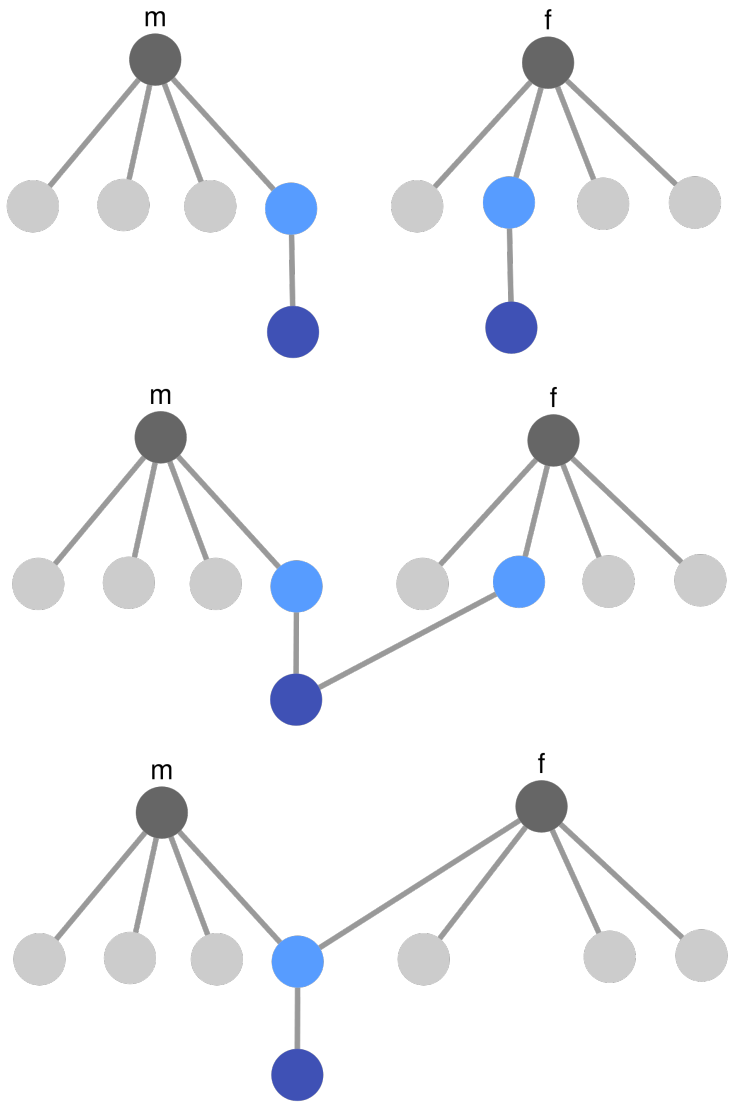
```
def indProof(l: List[Int]): Boolean = {
  if l.isEmpty then true
  else if !l.tail.isEmpty && l.head > l.tail.head then false
  else
    val t = l.tail
    if t.isEmpty then true
    else if !t.tail.isEmpty && t.head > t.tail.head then false
    else indProof(t.tail)
    assume(isSortedF(t) == isSortedM(t))
} ensuring(isSortedF(l) == isSortedM(l))
```

Function Call Matching

Function Call Matching

Function Call Matching: Solution

- Replace function calls by calls to equivalent functions



Function Call Matching: An Example

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

def uniqueF(l1: List[Int], l2: List[Int]): List[Int] =
  l2 match
    case Nil() => l1
    case Cons(hd, tl) =>
      if isInF(hd, l1)
      then uniqueF(l1, tl)
      else uniqueF(l1 ++ List[Int](hd), tl)

def uniqF(lst: List[Int]): List[Int] =
  uniqueF(Nil(), lst)
```

Example 2

- exercises/2_uniq.scala
- Task:
  Use the debug=trees option to observe the transformations

Function Call Matching: Type- and Test-Directed Search

```
def isInM(lst: List[Int], a: Int): Boolean
def uniqueM(l1: List[Int], l2: List[Int]): List[Int]
def uniqM(lst: List[Int]): List[Int]
```

```
def isInF(a: Int, lst: List[Int]): Boolean
def uniqueF(l1: List[Int], l2: List[Int]): List[Int]
def uniqF(lst: List[Int]): List[Int]
```

Function Call Matching: Type- and Test-Directed Search

```
def isInM(lst: List[Int], a: Int): Boolean
def uniqueM(l1: List[Int], l2: List[Int]): List[Int]
def uniqM(lst: List[Int]): List[Int]
```

```
def isInF(a: Int, lst: List[Int]): Boolean
def uniqueF(l1: List[Int], l2: List[Int]): List[Int]
def uniqF(lst: List[Int]): List[Int]
```

- Type-directed search
- Test-directed search

Function Call Matching: Type- and Test-Directed Search

```
def isInM(lst: List[Int], a: Int): Boolean
def uniqueM(l1: List[Int], l2: List[Int]): List[Int]
def uniqM(lst: List[Int]): List[Int]
```

```
def isInF(a: Int, lst: List[Int]): Boolean
def uniqueF(l1: List[Int], l2: List[Int]): List[Int]
def uniqF(lst: List[Int]): List[Int]
```

- Type-directed search
- Test-directed search

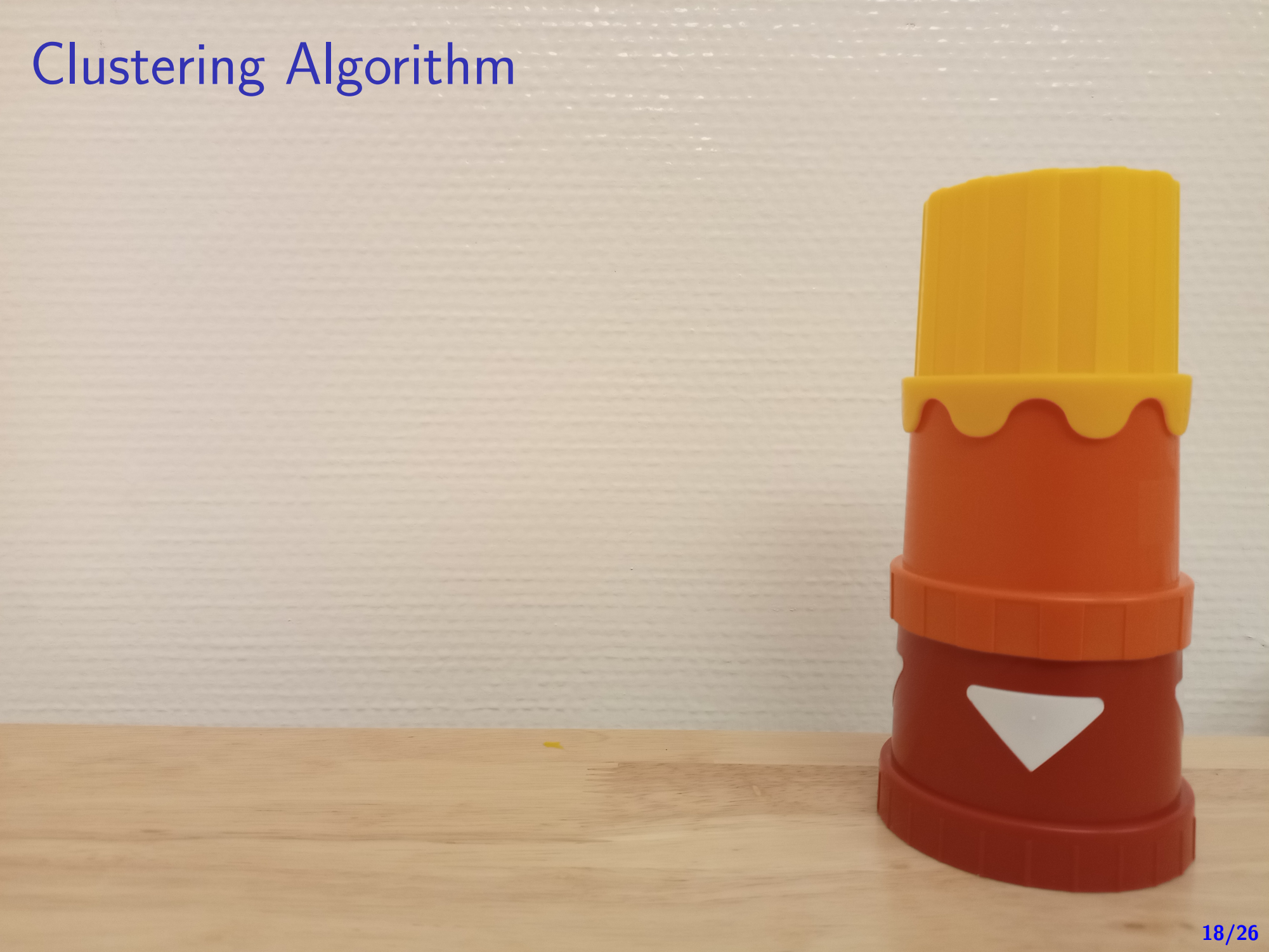
Function Call Matching: Substitutions

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

def uniqueF(l1: List[Int], l2: List[Int]): List[Int] =
  l2 match
    case Nil() => l1
    case Cons(hd, tl) =>
      if isInF(hd, l1) isInM(l1, hd)
      then uniqueF(l1, tl)
      else uniqueF(l1 ++ List[Int](hd), tl)

def uniqF(lst: List[Int]): List[Int] =
  uniqueF(Nil(), lst) uniqueM(lst, Nil())
```

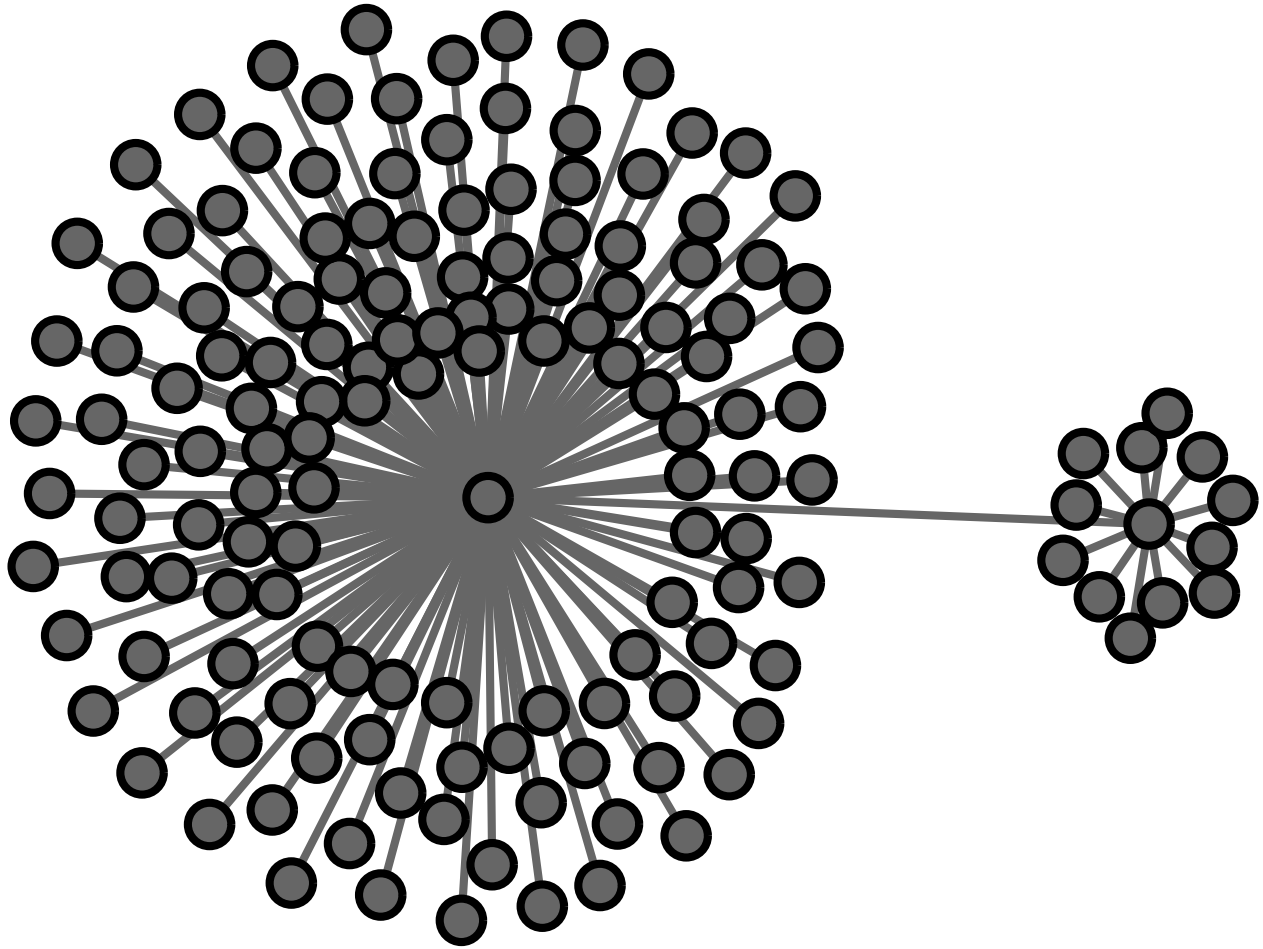
Clustering Algorithm



Clustering Algorithm

- Goal: Classify candidate programs
  using pairwise equivalence as a subroutine
- Input:
  Reference programs, candidate programs
- Output:
  Classification of candidate programs

Clustering Algorithm: Illustration



Example 3

- exercises/3_max.scala

Clustering Algorithm: Bridges

```
def maxM(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(h, Nil()) => h
  case Cons(h, t) => if h > maxM(t) then h else maxM(t)

def maxT(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(a, Nil()) => a
  case Cons(a, Cons(b, t)) =>
    if a > b then maxT(a :: t) else maxT(b :: t)

def maxF(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(h, t) =>
    t.foldLeft(h)((a, b) => if a >= b then a else b)
```

Clustering Algorithm: Bridges

```
def maxM(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(h, Nil()) => h
  case Cons(h, t) => if h > maxM(t) then h else maxM(t)

def maxT(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(a, Nil()) => a
  case Cons(a, Cons(b, t)) =>
    if a > b then maxT(a :: t) else maxT(b :: t)

def maxF(lst: List[Int]): Int = lst match
  case Nil() => Int.MinValue
  case Cons(h, t) =>
    t.foldLeft(h)((a, b) => if a >= b then a else b)
```

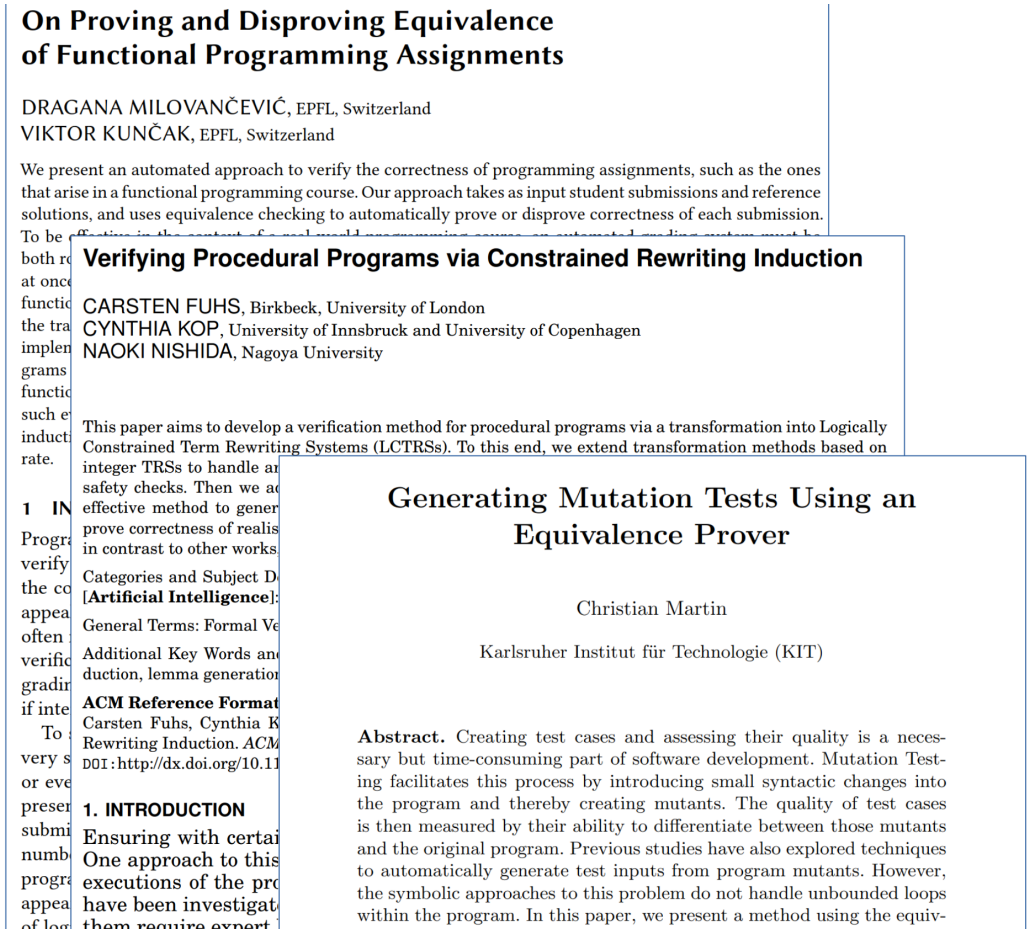
More Pieces

- Counterexamples
- Conditional equivalence
- Termination
- Performance
- Exceptions
- Loops

Equivalence Checking in Practice: Next Steps

- Digital Resources for Instruction and Learning (DRIL)

Ongoing and Suggested Projects



On 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.

Verifying Procedural Programs via Constrained Rewriting Induction

CARSTEN FUHS, Birkbeck, University of London
CYNTHIA KOP, University of Innsbruck and University of Copenhagen
NAOKI NISHIDA, Nagoya University

This paper aims to develop a verification method for procedural programs via a transformation into Logically Constrained Term Rewriting Systems (LCTRSs). To this end, we extend transformation methods based on integer TRSs to handle ar
safety checks. Then we ac
effective method to gener
prove correctness of realis
in contrast to other works.

Categories and Subject D
[**Artificial Intelligence**]:

General Terms: Formal Ve

Additional Key Words an
duction, lemma generatio

ACM Reference Format
Carsten Fuhs, Cynthia K
Rewriting Induction. *ACM*
DOI: http://dx.doi.org/10.11

1. INTRODUCTION

Ensuring with certai
One approach to this
executions of the pr
have been investigat
them require expert

Generating Mutation Tests Using an Equivalence Prover

Christian Martin

Karlsruher Institut für Technologie (KIT)

Abstract. Creating test cases and assessing their quality is a necessary but time-consuming part of software development. Mutation Testing facilitates this process by introducing small syntactic changes into the program and thereby creating mutants. The quality of test cases is then measured by their ability to differentiate between those mutants and the original program. Previous studies have also explored techniques to automatically generate test inputs from program mutants. However, the symbolic approaches to this problem do not handle unbounded loops within the program. In this paper, we present a method using the equiv-

Example 4

- exercises/4_find.scala

Example 4

- exercises/4_find.scala
- Homework:
  Prove the equivalence of exists1 and exists3

Resources

- epfl-lara.github.io/stainless/equivalence.html
- infoscience.epfl.ch/record/290689
- stainless/frontends/benchmarks/equivalence