Proving Termination via Measure Transfer in Equivalence Checking

Dragana Milovančević, Carsten Fuhs, Mario Bucev, Viktor Kunčak

November 13, 2024

iFM 2024

EPFL

Birkbeck UNIVERSITY OF LONDON

Why Equivalence Checking?

- Specification is an executable program, no need for pre- and post-conditions
- Potential for full automation, "push-button verification"
- Applications in regression verification, translation validation, high-performance computing, automated grading...

Why Equivalence Checking?

- Specification is an executable program, no need for pre- and post-conditions
- Potential for full automation, "push-button verification"
- Applications in regression verification, translation validation, high-performance computing, automated grading...

- Undecidable in general
- In the Stainless verifier: proofs by functional induction (also in Coq, Isabelle)
- Given two programs, automatically check program equivalence:
  - YES, if there is a proof ✓
  - NO, if there is a counterexample
  - TIMEOUT, otherwise

Equivalence Checking

- Warm-Up Example #1: Are uniq and uniq1 equivalent?

```
def uniq(lst: List[Int]): List[Int] =
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)

def uniq1(lst1: List[Int]): List[Int] =
  if lst1.isEmpty then Nil()
  else lst1.reverse match
    case Cons(hd, tl) =>
      if find(hd, tl) then uniq1(tl.reverse)
      else uniq1(tl.reverse) ++ List(hd)
```

Equivalence Checking

- Warm-Up Example #1: Are uniq and uniq1 equivalent?

```
def uniq(lst: List[Int]): List[Int] =
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)

def uniq1(lst1: List[Int]): List[Int] =
  if lst1.isEmpty then Nil()
  else lst1.reverse match
    case Cons(hd, tl) =>
      if find(hd, tl) then uniq1(tl.reverse)
      else uniq1(tl.reverse) ++ List(hd)
```

Yes. And Stainless can prove that they are equivalent! ✓

Equivalence Checking

- Warm-Up Example #2: Are uniq and uniq2 equivalent?

```
def uniq(lst: List[Int]): List[Int] =
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)

def uniq2(lst2: List[Int]): List[Int] =
  lst2.reverse match
    case Nil() => uniq2(lst2.reverse)
    case Cons(hd, tl) =>
      if find2(hd, tl) then uniq2(tl.reverse)
      else uniq2(tl.reverse) ++ List(hd)
```

Equivalence Checking

- Warm-Up Example #2: Are uniq and uniq2 equivalent?

```
def uniq(lst: List[Int]): List[Int] =
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)

def uniq2(lst2: List[Int]): List[Int] =
  lst2.reverse match
    case Nil() => uniq2(lst2.reverse)
    case Cons(hd, tl) =>
      if find2(hd, tl) then uniq2(tl.reverse)
      else uniq2(tl.reverse) ++ List(hd)
```

No. But Stainless can prove that they are equivalent! ✓ ?

Equivalence Checking

- Warm-Up Example #2: Are uniq and uniq2 equivalent?

```
def uniq(lst: List[Int]): List[Int] =
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)

def uniq2(lst2: List[Int]): List[Int] =
  lst2.reverse match
    case Nil() => uniq2(lst2.reverse)
    case Cons(hd, tl) =>
      if find2(hd, tl) then uniq2(tl.reverse)
      else uniq2(tl.reverse) ++ List(hd)
```

Proving Termination

- In Stainless: termination measures (ranking functions)
- Also in Dafny, Why3, KeY...
- Manual decreases annotations (or, if we are lucky, inferred automatically)

Proving Termination

- In Stainless: termination measures (ranking functions)
- Also in Dafny, Why3, KeY...
- Manual decreases annotations (or, if we are lucky, inferred automatically)

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

def uniq(lst: List[Int]): List[Int] =
  decreases(lst.size) // user provides
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)
```

This program terminates! ✓

Proving Termination

- In Stainless: termination measures (ranking functions)
- Also in Dafny, Why3, KeY...
- Manual decreases annotations (or, if we are lucky, inferred automatically)

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

def uniq(lst: List[Int]): List[Int] =
  decreases(lst.size) // user provides
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)
```

This program terminates! ✓

- What about uniq1? What about uniq2?

Proving Termination: How does this scale?

- Manual annotations: takes a lot of time
- Automated measure inference: also takes time, and does not always work

```
def f(x: BigInt, p: BigInt => Boolean): BigInt =
  require(!p(x) || exists[BigInt]((j: BigInt) => j < x && maxNegP(j, p)))
  decreases(if !p(x) then BigInt(0)
            else x - elim_exists[BigInt]((j: BigInt) => j < x && maxNegP(j, p)))
  if p(x) then f(x - 1, p)
  else x
```

github.com/epfl-lara/stainless/blob/main/frontends/benchmarks/termination/valid/Partial.scala

Measure Transfer

- Heuristic: Equivalent programs terminate for the same reason

```
def uniq(lst: List[Int]): List[Int] =
  decreases(lst.size) // user provides
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)
```

Measure Transfer

- Heuristic: Equivalent programs terminate for the same reason

```
def uniq(lst: List[Int]): List[Int] =
  decreases(lst.size) // user provides
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)

def uniq1(lst1: List[Int]): List[Int] =
  if lst1.isEmpty then Nil()
  else lst1.reverse match
    case Cons(hd, tl) =>
      if find(hd, tl) then uniq1(tl.reverse)
      else uniq1(tl.reverse) ++ List(hd)
```

Measure Transfer

- Heuristic: Equivalent programs terminate for the same reason

```
def uniq(lst: List[Int]): List[Int] =
  decreases(lst.size) // user provides
  lst.reverse match
    case Nil() => Nil()
    case Cons(hd, tl) =>
      if !find(hd, tl) then uniq(tl.reverse) :+ hd
      else uniq(tl.reverse)

def uniq1(lst1: List[Int]): List[Int] =
  decreases(lst11.size) // inferred by measure transfer
  if lst1.isEmpty then Nil()
  else lst1.reverse match
    case Cons(hd, tl) =>
      if find(hd, tl) then uniq1(tl.reverse)
      else uniq1(tl.reverse) ++ List(hd)
```

Measure Transfer

- Heuristic: Equivalent programs terminate for the same reason

```
def f(x: BigInt, p: BigInt => Boolean): BigInt =
  require(!p(x) || exists[BigInt]((j: BigInt) => j < x && maxNegP(j, p)))
  decreases(if !p(x) then BigInt(0)
            else x - elim_exists[BigInt]((j: BigInt) => j < x && maxNegP(j, p)))
  if p(x) then f(x - 1, p)
  else x
```

Measure Transfer

- Heuristic: Equivalent programs terminate for the same reason

```
def f(x: BigInt, p: BigInt => Boolean): BigInt =
  require(!p(x) || exists[BigInt]((j: BigInt) => j < x && maxNegP(j, p)))
  decreases(if !p(x) then BigInt(0)
            else x - elim_exists[BigInt]((j: BigInt) => j < x && maxNegP(j, p)))
  if p(x) then f(x - 1, p)
  else x

def f1(q: BigInt => Boolean, y: BigInt): BigInt =
  require(!q(y) || exists[BigInt]((j: BigInt) => j < y && maxNegP(j, q)))
  if q(y) then f1(q, y - 1)
  else y
```

Measure Transfer

- Heuristic: Equivalent programs terminate for the same reason

```
def f(x: BigInt, p: BigInt => Boolean): BigInt =
  require(!p(x) || exists[BigInt]((j: BigInt) => j < x && maxNegP(j, p)))
  decreases(if !p(x) then BigInt(0)
            else x - elim_exists[BigInt]((j: BigInt) => j < x && maxNegP(j, p)))
  if p(x) then f(x - 1, p)
  else x

def f1(q: BigInt => Boolean, y: BigInt): BigInt =
  require(!q(y) || exists[BigInt]((j: BigInt) => j < y && maxNegP(j, q)))
  decreases(if !q(y) then BigInt(0)
            else y - elim_exists[BigInt]((j: BigInt) => j < y && maxNegP(j, q)))
  if q(y) then f1(q, y - 1)
  else y
```

Measure Transfer in Program Refactoring

Name	LOC	F	D	I	IT[s]	T	TT[s]
AdjList	32	2	1	✓	26.12	✓	23.74
ArrayContent	12	1	1	✓	12.85	✓	13.32
ArrayHeap	58	4	1	✓	27.47	✓	26.19
ArrayInc	15	2	1	✗	N/A	✓	18.12
Boardgame	293	8	3	✗	N/A	✓	1186.6
FiniteStreams	28	1	1	✗	N/A	✓	16.07
MaxHeapify	51	3	1	✗	N/A	✓	15.12
Partial	27	4	1	✗	N/A	✓	15.80
SortedArray	26	2	1	✓	15.18	✓	13.03
Valid2DLen	17	1	1	✗	N/A	✓	19.10

- IT / TT: time for equivalence checking + measure inference / transfer

Measure Transfer in Programming Assignments

Name	LOC	F	D	R	S	I	T
gcd	9	1	1	2	41	0	22
formula	59	2	1	1	37	0	27
prime	21	4	2	2	22	0	5
sigma	10	1	1	3	704	0	678

Resources: Implementation, Benchmarks

- Implementation on top of the Stainless verifier for Scala
- Open source: github.com/epfl-lara/stainless
- Accompanying artifact with all our benchmarks: zenodo.org/records/13787855

Conclusions

- Proving program termination can be challenging and time consuming
- Measure transfer leads to improvements in automation and applicability
- Future work: consider more complex measure transformations, such as transfer of auxiliary termination lemmas

Stainless: github.com/epfl-lara/stainless
iFM'24 Artifact: zenodo.org/records/13787855