Proving Termination of Scala Programs by Constrained Term Rewriting

Dragana Milovančević, Carsten Fuhs, Viktor Kunčak

July 20, 2025
WPTE'25, Birmingham, UK

Motivation: Termination Proofs for Program Equivalence

- Warm-Up Example: Are f and f2 equivalent?

```
def f(t: Formula): Boolean = t match
  case True => true
  case False => false
  case Not(n) => if f(n) then false else true
  case Imply(l, r) => f(Not(l)) || f(r)

def f2(t: Formula): Boolean = t match
  case True => true
  case False => false
  case Not(n) => !f2(t)
  case Imply(l, r) => f2(Not(l)) || f2(r)
```

Motivation: Termination Proofs for Program Equivalence

- Warm-Up Example: Are f and f2 equivalent?

```
def f(t: Formula): Boolean = t match
  case True => true
  case False => false
  case Not(n) => if f(n) then false else true
  case Imply(l, r) => f(Not(l)) || f(r)

def f2(t: Formula): Boolean = t match
  case True => true
  case False => false
  case Not(n) => !f2(t)
  case Imply(l, r) => f2(Not(l)) || f2(r)
```

No. But, without termination checks, tools can claim otherwise!

Motivation: Termination Proofs for Program Equivalence

- Warm-Up Example: Are f and f2 equivalent?

```
def f(t: Formula): Boolean = t match
  case True => true
  case False => false
  case Not(n) => if f(n) then false else true
  case Imply(l, r) => f(Not(l)) || f(r)

def f2(t: Formula): Boolean = t match
  case True => true
  case False => false
  case Not(n) => !f2(t)
  case Imply(l, r) => f2(Not(l)) || f2(r)
```

No. But, without termination checks, tools can claim otherwise!

Context: Stainless Scala Verifier

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

Termination Proving in Stainless

- Termination measures (ranking functions)
- Not only in Stainless: also in Dafny, Why3, KeY...
- Manual decreases annotations (or, if we are lucky, inferred automatically)

Termination Proving in Stainless

- Termination measures (ranking functions)
- Not only in Stainless: also in Dafny, Why3, KeY...
- Manual decreases annotations (or, if we are lucky, inferred automatically)

```
def f(t: Formula): Boolean =
  decreases(t.size) // user provides
  t match
    case True => true
    case False => false
    case Not(n) => if f(n) then false else true
    case Imply(l, r) => f(Not(l)) || f(r)
```

This program terminates! ✓

How does this scale?

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

Bonus Background: Measure Transfer in Stainless

- Heuristic: Equivalent programs terminate for the same reason
- + Improves the scalability over measure inference
- – Still requires the initial measure, manual effort

Our Work: Proving Termination by Constrained Term Rewriting

- Goal: Use termination provers that operate on term rewrite systems
- Already used for proving termination of programs in Haskell, Java, Prolog, C
- AProVE, Cora, Ctrl: use logically constrained term rewrite systems (LCTRS)

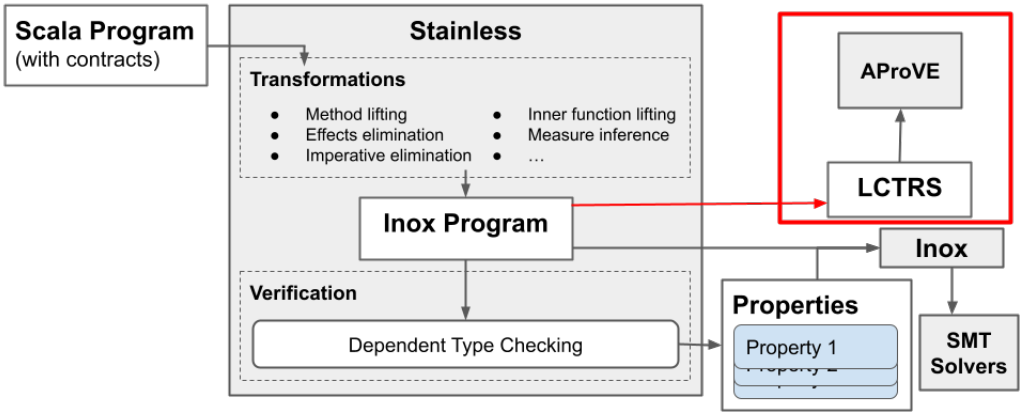
```
(VAR n l r tmp1 tmp2)
(RULES
  f(True) -> TRUE
  f(False) -> FALSE
  f(Not(n)) -> f1(Not(n), f(n))
  f1(Not(n), tmp1) -> FALSE :|: tmp1
  f1(Not(n), tmp1) -> TRUE :|: !tmp1
  f(Imply (l,r)) -> f2(Imply (l,r), f(Not(l)))
  f2(Imply (l,r), tmp2) -> TRUE :|: tmp2
  f2(Imply (l,r), tmp2) -> f(r) :|: !tmp2
)
```

Termination was proven ✓

From Scala to LCTRS

- Input: Stainless (Inox) trees
- Output: LCTRS rules
  - Currently, AProVE's Integer Term Rewrite System syntax
- Goal: LCTRS termination implies Scala program termination

Stainless Pipeline, Including Our Modifications (in red)



Example: Translating Specifications

```
def gcd(a: Int, b: Int): Int = {
  require(a >= 0 && b >= 0)
  if b == 0 then a else gcd(b, a%b)
} ensuring(true)
```

```
gcd(a,b) -> gcd0(a,b)
gcd0(a,b) -> gcd1(a,b,b = 0) :|: a >= 0 /\ b >= 0
gcd1(a,b,tb) -> gcd2(a,b,tb) :|: tb /\ a >= 0 /\ b >= 0
gcd1(a,b,tb) -> gcd3(a,b,tb) :|: !tb /\ a >= 0 /\ b >= 0
gcd2(a,b,tb) -> gcd6(a,b,a) :|: tb /\ a >= 0 /\ b >= 0
gcd3(a,b,tb) -> gcd4(a,b,tb,a % b) :|: !tb /\ a >= 0 /\ b >= 0
gcd4(a,b,tb,tmp0) -> gcd5(a,b,tb,tmp0,gcd(b,tmp0)) :|: !tb /\ a >= 0 /\ b >= 0
gcd5(a,b,tb,tmp0,ret_gcd(fresh)) -> gcd6(a,b,fresh) :|: !tb /\ a >= 0 /\ b >= 0
gcd6(a,b,res) -> gcd7(a,b,res,res) :|: TRUE /\ a >= 0 /\ b >= 0
gcd7(tmp1,tmp2,tmp3,ret0) -> ret_gcd(ret0)
```

Evaluation on Programming Assignments

Name	LOC	F	D	R	S	Inference	Transfer	LCTRS	Total Proven
formula	59	2	1	1	37	0	27	24	28
sigma	10	1	1	3	704	0	678	0	678
prime	21	4	2	2	22	0	5	14	14
gcd	9	1	1	2	41	0	22	15	27

Discussion

- Unbounded vs bounded Scala integers

```
def overflow_fun(i: Int, n: Int): Int =
  if i <= n then overflow_fun(i + 1, n) else i
```

Discussion

- Unbounded vs bounded Scala integers

```
def overflow_fun(i: Int, n: Int): Int =
  if i <= n then overflow_fun(i + 1, n) else i
```

- Limited support for laziness (e.g. short-circuit evaluation); no lazy vals

Discussion

- Unbounded vs bounded Scala integers

```
def overflow_fun(i: Int, n: Int): Int =
  if i <= n then overflow_fun(i + 1, n) else i
```

- Limited support for laziness (e.g. short-circuit evaluation); no lazy vals
- Analysis for arbitrary terms; requires termination of all functions

```
def main(x: BigInt, y: BigInt): BigInt =
  require(x > y)
  helper(x, y)

def helper(x: BigInt, y: BigInt): BigInt =
  if x <= 0 then y - x
  else if 2 * x > y then x - y
  else 1 + helper(x, y)
```

This program does not terminate (!) ✗

Conclusions

- Automated approach for termination analysis of Scala programs via a translation to LCTRSs
- Integrated into the transformation pipeline of the Stainless Scala verifier
- Preliminary experiments show an improvement over existing techniques in Stainless
  - Potential for a hybrid approach combined with measure transfer

Future Work

- Add support for more Scala features (primarily higher-order functions)
- Add support for more tools as an alternative to AProVE
- Prove correctness of Scala to LCTRS translation
- Propagate information from the termination analysis to Scala