Proving Termination of Scala Programs by Constrained Term Rewriting

Dragana Milovančević
EPFL, Station 14
CH-1015 Lausanne, Switzerland
dragana.milovancevic@epfl.ch

Carsten Fuhs
Birkbeck, University of London
United Kingdom
c.fuhs@bbk.ac.uk

Viktor Kunčak
EPFL, Station 14
CH-1015 Lausanne, Switzerland
viktor.kuncak@epfl.ch

Techniques and tools for program verification often require a termination proof as an ingredient of correctness proofs for other program properties (e.g., program equivalence). However, the range of termination proof techniques is vast, so implementing them for every single programming language anew would not be an efficient use of time for tool developers.

In this extended abstract, we propose an indirect route for proving termination of Scala programs, via a translation to constrained term rewriting systems so that termination of the rewrite system implies termination of the Scala program. This allows for separating the language-specific and the language-independent aspects of the analysis into different tools. We have implemented our approach in the Stainless program verifier. Preliminary experiments with the termination prover AProVE indicate that this approach can lead to an increased termination success rate compared to existing approaches.

1 Introduction

One of the tasks for program analysis tools is to prove termination of their input programs. This task matters both on its own and also as a precondition for proof techniques for other properties, such as program equivalence [20] and properties expressed using multiple user-defined function invocations. While implementing language-specific proof techniques is an option, it is arduous and ignores opportunities for reuse of existing infrastructure: over the last decades, techniques and tools for push-button termination analysis of different computational formalisms have matured, particularly so for various flavors of *Term Rewriting Systems (TRSs)*. This suggests a two-stage approach to termination analysis by program analysis tools with a separation of concerns: (1) extract a TRS from the input program such that termination of the TRS implies termination of the input program, and (2) call an external termination tool for TRSs to get a termination proof (if one can be found).

Similar two-stage approaches for analysis of properties via term rewriting have been used in the literature for proofs of termination of programs in many languages (Haskell [8], Java [25], Prolog [9,27], and C [6,15]), for proofs of equivalence (e.g., in C programs [6,15]), or inference of complexity bounds (for Prolog [9], Jinja [22], and OCaml [1]).

A strength of term rewriting is that it can express inductive data types in a natural way. However, classic term rewriting has a drawback for this use in program analysis: it does not offer a direct representation of primitive data types, which are predefined in most programming languages (e.g., integer numbers and their operations). As classic term rewriting is Turing-complete, it is possible to encode these data types and their operations via terms and recursive rewrite rules. However, a lot of the available domain knowledge baked into the programming language semantics (e.g., the meaning of the operators such as $+$ and $*$) would be "obfuscated in translation", so the existing infrastructure for automated reasoning, such as SMT solvers, cannot directly benefit from this domain knowledge, e.g., in the search for

```
expr ::=
  literal
  | id
  | expr ~ expr
  | not expr
  | expr.id
  | id(expr, ..., expr)
  | val id = expr in expr
  | if expr then expr else expr
  | if expr isInstanceOf id then expr else expr
  | assume(expr) in expr

literal ::=
  integerLiteral
  | true
  | false
```

Figure 1: Input syntax, where $\sim$ is a symbol in $\{+, -, *, /, \%, \text{and}, \text{or}, >, \geq, <, \leq, ==\}$.

```
rule ::=
  term → term
  | term → term ⟨ term ⟩


term ::=
  literal
  | id
  | id ( term* )
  | term ~ term
  | ! term

literal ::=
  integerLiteral
  | TRUE
  | FALSE
```

Figure 2: LCTRS syntax, where $\sim$ is a symbol in $\{+, -, *, /, \%, \text{and}, \text{or}, >, \geq, <, \leq, ==\}$.

a termination proof step. This is why in recent years, *constrained* term rewriting [4, 11, 13] has gained popularity for applications in program analysis. This form of term rewriting is augmented with built-in logical constraints, which can be analyzed by SMT solvers. Kop and Nishida [13] proposed *Logically Constrained Term Rewriting Systems (LCTRSs)* as an attempt at unifying different forms of constrained term rewriting. This formalism has gained traction in the community, and several research groups have built tools to analyze different properties of LCTRSs (e.g., Cora [16], CRaris [23], crest [28], Ctrl [14], RMT [2], TcT [32]).

In this work, we propose a translation from Scala programs to LCTRSs such that termination of the generated LCTRS implies termination of the original Scala program. We have implemented the translation in the Stainless program verifier [17]. Stainless already contains a pipeline that successively "desugars" Scala programs for their analysis and applies program transformations between different tree representations of the program. Our prototype implementation enters the tool chain at a suitable level, allowing Stainless to extract an LCTRS for delegation of the termination analysis to an external tool.

To analyze termination of the extracted LCTRSs, our implementation calls the program analysis tool AProVE [7]. As Scala (more precisely, the language fragment that our translation supports) uses a call-by-value evaluation strategy, a proof of *innermost* termination of the generated LCTRS suffices to prove termination of the original Scala program. In this extended abstract, we present our Scala-to-LCTRS export and a preliminary evaluation.

2 Language and Background

We use LCTRSs [13] as an intermediate representation for proving termination of Scala [24] programs. This section defines the subset of Scala that we consider and gives a brief background on LCTRSs.

We consider a purely functional subset of Scala. The input to our system is an abstract syntax tree (AST) that we extract from Stainless at the last stage of its transformation pipeline. We define the supported AST syntax in Figure 1.

An LCTRS is defined as a sequence of rules $l \to r \langle\ C\ \rangle$, where l and r are terms, and C is a logical constraint. We define the supported LCTRS syntax in Figure 2. Termination of the LCTRSs that we generate can be checked by external tools such as AProVE [7], which we used in our evaluation, Ctrl [14], or Cora [16]. Initial experimentation with the three tools indicated that AProVE was the most successful in dealing with systems originating from automated translations, which tend to have many more rewrite rules than a hand-optimized system, as we illustrate on an example in the next section. Our translation from Scala to LCTRSs is inspired by the related work on converting C programs to LCTRS in the c2lctrs tool [15]. After illustrating our translation on an example Scala program in Section 3, we define the translation rules in Section 4.

3 Illustrative Example: Formula Programming Assignment

This section gives an overview of our Stainless to LCTRS export on one illustrative example.

We consider the example student submission from the formula benchmark in related work on automated grading of programming assignments [20,29]. Specifically, we consider a minimized definition to focus on the essence of our translation:

```
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

In our evaluation in Section 5, we consider the full definition from the formula benchmark, which includes cases for boolean variables, conjunction and disjunction.

Figure 3 shows the Scala program under analysis. This example is particularly challenging for termination proving due to the recursive call f(Not(l)), whose argument does not trivially decrease over Imply(l, r). In Stainless, measure inference for functions on algebraic data types considers the structural size of data type arguments. In our example, the default size function defines the size of Imply(True, True) as 1 + 0 + 0 = 1, and the size of Not(True) as 1 + 0 = 1 (not decreasing). As a result, Stainless fails to automatically infer a termination measure for this function and cannot prove termination. We thus turn to external termination provers.

The remainder of Figure 3 shows the input of our translation (Stainless tree as defined in Figure 1) and the output of our translation (LCTRS rules as defined in Figure 2). Our translation traverses the input Stainless trees and iteratively constructs the corresponding LCTRS rules. To capture the semantics of the program control flow, during the translation we keep track of conditions as constraints along the execution path. For example, the **else** branch from **if** f(Not(s.l)) **then** true **else** f(s.r) produces the following condition:

```
f16(t, Imply(s_l, s_r), tmp1) → f18(t, Imply(s_l, s_r), tmp1) ⟨ !tmp1 ⟩
```

which is then preserved in the subsequent rules for this **else** branch (symbols f18 to f22).

The AProVE termination prover successfully proves termination of our resulting LCTRS, ensuring termination of the original Scala program.

This simplified example already illustrates interesting aspects of our translation, such as handling of algebraic data types (ADTs) and ADT selectors.

```
// Scala source code
def f(t: Formula): Boolean = t match
  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

```
// Intermediate representation of source code
def f(t: Formula): Boolean =
  val s: Formula = t
  if s isInstanceOf True then true
  else if s isInstanceOf False then false
  else if s isInstanceOf Not then
    val tb: Boolean = f(s.p)
    if tb then false else true
  else if s isInstanceOf Imply then f(Not(s.l)) || f(s.r)
```

```
f(t) → f1(t, t)
f1(t, True) → f2(t, True)
f1(t, Not(s_p)) → f3(t, Not(s_p))
f1(t, False) → f3(t, False)
f1(t, Imply(s_l, s_r)) → f3(t, Imply(s_l, s_r))
f2(t, True) → f22(t, True, TRUE)
f3(t, False) → f4(t, False)
f3(t, Not(s_p)) → f5(t, Not(s_p))
f3(t, True) → f5(t, True)
f3(t, Imply(s_l, s_r)) → f5(t, Imply(s_l, s_r))
f4(t, False) → f22(t, False, FALSE)
f5(t, Not(s_p)) → f6(t, Not(s_p))
f5(t, True) → f12(t, True)
f5(t, False) → f12(t, False)
f5(t, Imply(s_l, s_r)) → f12(t, Imply(s_l, s_r))
f6(t, Not(s_p)) → f7(t, Not(s_p), s_p)
f7(t, Not(s_p), tmp0) → f8(t, Not(s_p), tmp0, f(tmp0))
f8(t, Not(s_p), tmp0, r_f(w)) → f9(t, Not(s_p), w)
f9(t, Not(s_p), tb) → f10(t, Not(s_p), tb) ⟨ tb ⟩
f9(t, Not(s_p), tb) → f11(t, Not(s_p), tb) ⟨ !tb ⟩
f10(t, Not(s_p), tb) → f22(t, Not(s_p), FALSE) ⟨ tb ⟩
f11(t, Not(s_p), tb) → f22(t, Not(s_p), TRUE) ⟨ !tb ⟩
f12(t, Imply(s_l, s_r)) → f13(t, Imply(s_l, s_r))
f12(t, Not(s_p)) → f21(t, Not(s_p))
f12(t, True) → f21(t, True)
f12(t, False) → f21(t, False)
f13(t, Imply(s_l, s_r)) → f14(t, Imply(s_l, s_r), Not(s_l))
f14(t, Imply(s_l, s_r), tmp2) → f15(t, Imply(s_l, s_r), tmp2, f(tmp2))
f15(t, Imply(s_l, s_r), tmp2, r_f(w)) → f16(t, Imply(s_l, s_r), w)
f16(t, Imply(s_l, s_r), tmp1) → f17(t, Imply(s_l, s_r), tmp1) ⟨ tmp1 ⟩
f16(t, Imply(s_l, s_r), tmp1) → f18(t, Imply(s_l, s_r), tmp1) ⟨ !tmp1 ⟩
f17(t, Imply(s_l, s_r), tmp1) → f22(t, Imply(s_l, s_r), TRUE) ⟨ tmp1 ⟩
f18(t, Imply(s_l, s_r), tmp1) → f19(t, Imply(s_l, s_r), tmp1, s_r) ⟨ !tmp1 ⟩
f19(t, Imply(s_l, s_r), tmp1, tmp3) → f20(t, Imply(s_l, s_r), tmp1, tmp3, f(tmp3)) ⟨ !tmp1 ⟩
f20(t, Imply(s_l, s_r), tmp1, tmp3, r_f(w)) → f22(t, Imply(s_l, s_r), w) ⟨ !tmp1 ⟩
f21(t, s) → f22(t, s, e)
f22(tmp4, tmp5, ret) → r_f(ret)
```

Figure 3: Input to Stainless: Scala program (top-left). Input to our translation: Stainless tree at the last pipeline stage, after pattern matching elimination (top-right). Output of our translation: LCTRS rules (bottom). The ⟨ ⟩ syntax denotes the constraints gathered along the execution path.

4 Scala to LCTRS Translation

In this section, we present our translation from Scala to LCTRS.

We take a Scala program as an abstract syntax tree at the last stage of the Stainless pipeline (Inox trees). Figure 1 illustrates the syntax of expressions in these input trees. We then perform a series of transformations to obtain the output trees corresponding to a LCTRS. Figure 2 illustrates the syntax of these output trees. Our pretty-printer then exports the resulting LCTRS rules such that their termination with regard to innermost rewriting can be analyzed by AProVE.

We define the series of transformations as follows:

1. Pre-transformation phase
2. Translation phase
3. Post-transformation phase.

The pre-transformation phase consists of let-transformations (introducing let bindings for expressions) and short-circuiting (eliminating and/or operators by transforming them to if-then-else). For example, in our formula submission in Section 3, the boolean expression f(Not(s.l)) || f(s.r) is transformed to:

```
val tmp1 = f(Not(s.l))
if tmp1 then true else f(s.r),
```

which produces rules corresponding to the symbols f14 to f22 in Figure 3.

The post-transformation phase consists of introducing function wrappers, that is, rules that simplify extraction of function results. For example, in our formula submission in Figure 3, the symbol r_f enables the extraction of function f's result.

The main translation phase is defined in the convert function, which takes the following input:

– f, the name of the function under conversion;
– i, the number of symbols already declared for this function;
– $\overrightarrow{x} = [x_1, ..., x_n]$, a sequence of variables which are known at the start of the function or block;
– $\overrightarrow{y} = [y_1, ..., y_m]$, a sequence of locally declared variables (to be erased at the end of the function or block);
– R: the set of rules declared so far;
– C: the constraints so far;
– S: the statement to convert.

It produces the following output:

– R': the resulting set of rules (the original set R union rules from converting the S statement);
– j: the last index of symbols declared for the S statement.

At the end of this conversion, the resulting set R' defines the resulting LCTRS. To translate a full program, we convert each individual function and then take the union of the resulting sets R' for each function.

We next consider the definition of convert for each case in Figure 1. We use the syntax $\overrightarrow{x}$ to denote a sequence of elements $x_1, ..., x_n$ and the ::: symbol to denote sequence concatenation. We denote substitution of t by b in a by $a\,[t := b\,]$. In each invocation of convert, Stainless expressions are highlighted in gray.

4.1 Assume Statements

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, assume(p) in s) := convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, $p \wedge C$, s).

4.2 Constants and Variables

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, a) := (R', i +1), where:

- $R' = R \cup \{\ f_i(\overrightarrow{x}, \overrightarrow{y}) \rightarrow f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, a)\ \langle\ C\ \rangle\}$

4.3 Binary Operators

For each binary operator $\sim$[1], we define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, $a \sim b$) := (R', i +1), where:

- $R' = R \cup \{\ f_i(\overrightarrow{x}, \overrightarrow{y}) \rightarrow f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, a \sim b)\ \langle\ C\ \rangle\}$

4.4 Not

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, not a) := (R', i +1), where:

- $R' = R \cup \{\ f_i(\overrightarrow{x}, \overrightarrow{y}) \rightarrow f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, \text{not } a)\ \langle\ C\ \rangle\}$

4.5 Function Invocations

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, $g(e_1,\ldots,e_n)$) := (R', i +2), where:

- $R' = R \cup \{\ f_i(\overrightarrow{x}, \overrightarrow{y}) \rightarrow f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, g(e_1,\ldots,e_n))\ \langle\ C\ \rangle,$
  $f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, r_g(t)) \rightarrow f_{i+2}(\overrightarrow{x}, \overrightarrow{y}, t)\ \langle\ C\ \rangle\}$

Here, the term $r_g(t)$ denotes extraction of function g's result. This term is introduced in the post-transformation phase, as the right-hand side of the last rule in g's translation. For example, in the formula submission in Figure 3, the function call f(r) evaluates to the term r_f(w) below the symbol f20.

4.6 Let Bindings

To convert a statement val $b = d$ in e:

- Let (R_1, k) be the result of convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, {}, C, d)
- Let R_1' be R_1 $[f_k(e_1,\ldots,e_m)\ := f_k(e_1,\ldots,e_{|\vec{x}|+|\vec{y}|}, e_m)]$
- Let R_2 be $R \cup R_1$'

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, val $b = d$ in e) := convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$ ::: [b], R_2, C, e).

Here, the sequence $e_1,\ldots,e_{|\vec{x}|+|\vec{y}|}, e_m$ denotes the removal of local variables which are out of scope at the end of a block. The same substitution takes place in the branches of the **if** expressions (Section 4.7) and in pattern matching cases (Section 4.9).

4.7 If Expressions

To convert a statement if c then s else t:

- Let (R_2, k) be the result of convert(f, i +1, $\overrightarrow{x}$ ::: $\overrightarrow{y}$, [], {}, $c \wedge C$, s)
- Let (R_3, n) be the result of convert(f, k, $\overrightarrow{x}$ ::: $\overrightarrow{y}$, [], {}, $!c \wedge C$, t)
- Let R_2' be R_2 $[f_k(e_1,\ldots,e_m)\ := f_n(e_1,\ldots,e_{|\vec{x}|+|\vec{y}|}, e_m)]$
- Let R_3' be R_3 $[f_n(e_1,\ldots,e_m)\ := f_n(e_1,\ldots,e_{|\vec{x}|+|\vec{y}|}, e_m)]$

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, if c then s else t) := (R', n), where:

- $R' = R \cup \{f_i(\overrightarrow{x}, \overrightarrow{y}) \rightarrow f_{i+1}(\overrightarrow{x}, \overrightarrow{y})\ \langle\ c \wedge C\ \rangle, f_i(\overrightarrow{x}, \overrightarrow{y}) \rightarrow f_k(\overrightarrow{x}, \overrightarrow{y})\ \langle\ !c \wedge C\ \rangle\} \cup R_2' \cup R_3'$

For example, a function

[1]At this stage of the translation, $\sim$ can be any of the symbols in $\{+, -, *, /, \%, >, \geq, <, \leq, ==\}$. The symbols and, or cannot appear because they get eliminated in the pre-transformation phase.

```
def foo(x: BigInt): BigInt =
  if x > 0 then g(x) else h(x)
```

would be translated to:

```
foo0(x) → foo1(x, x)
foo1(x, tmp5) → foo2(x, tmp5, 0)
foo2(x, tmp5, tmp6) → foo3(x, tmp5 > tmp6)
foo3(x, tb) → foo4(x, tb) ⟨ tb ⟩
foo3(x, tb) → foo6(x, tb) ⟨ !tb ⟩
foo4(x, tb) → foo5(x, tb, g(x)) ⟨ tb ⟩
foo5(x, tb, ret_g(fresh0)) → foo8(x, tb, fresh0) ⟨ tb ⟩
foo6(x, tb) → foo7(x, tb, h(x)) ⟨ !tb ⟩
foo7(x, tb, ret_h(fresh1)) → foo8(x, tb, fresh1) ⟨ !tb ⟩
foo8(tmp7, tmp8, ret1) → ret_foo(ret1)
```

This example shows how the control flow of an **if** expression is represented with the help of different function symbols foo4 and foo6 such that a rewrite sequence starting from foo(t) for an integer t will result in exactly one of the branches of the **if** expression being evaluated.

In contrast, with a translation via rules

```
foo(x) → myif(x > 0, foo1(x), foo2(x))
foo1(x) → r_g(g(x))
foo2(x) → r_h(h(x))
myif(TRUE, x, y) → x
myif(FALSE, x, y) → y
```

with a single function symbol myif for all **if** expressions, both branches would be evaluated (which is why we avoid this alternative translation).

4.8 ADT Selectors (Field Accesses)

To convert a statement $a.e_k$:

– Let $\overrightarrow{xy}$' be ($\vec{x}$ ::: $\vec{y}$)[$a := A$ ($\vec{w}$)], where A is the constructor of $a.e_k$ and $\vec{w}$ fresh field variables.

We define convert(f, i, $\vec{x}$, $\vec{y}$, R, C, $a.e_k$) := (R', i +1), where:

– R' = $R \cup \{ f_i(\overrightarrow{xy}') \to f_{i+1}(\vec{x}, \vec{y}, w_k) \langle C \rangle \}$

4.9 Pattern Matching

To convert a statement if a isInstanceOf A_l then s else t:

– Let { A_1, ..., A_m } be all constructors of the supertype of A_l

– Let $\overrightarrow{xy_1}$, ... , $\overrightarrow{xy_m}$ be ($\vec{x}$ ::: $\vec{y}$)[$a := A_1$ ($\vec{u}$)], ..., ($\vec{x}$ ::: $\vec{y}$)[$a := A_m$ ($\vec{v}$)], respectively

– Let (R_2, k) be the result of convert(f, i +1, $\overrightarrow{xy_l}$, [], {}, C, s)

– Let (R_3, n) be the result of convert(f, k, $\vec{x}$ ::: $\vec{y}$, [], {}, C, t)

– Let R_2' be R_2 [$f_k(e_1,\dots,e_m)$:= $f_n(e_1,\dots,e_{|\vec{x}|+|\vec{y}|},e_m)$]

We define convert(f, i, $\vec{x}$, $\vec{y}$, R, C, if a isInstanceOf A_l then s else t) := (R', n), where:

– R' = $R \cup \{$
$f_i(\overrightarrow{xy_1}) \to f_k(\overrightarrow{xy_1}) \langle C \rangle$,..., $f_i(\overrightarrow{xy_{l-1}}) \to f_k(\overrightarrow{xy_{l-1}}) \langle C \rangle$,
$f_i(\overrightarrow{xy_l}) \to f_{i+1}(\overrightarrow{xy_l}) \langle C \rangle$,
$f_i(\overrightarrow{xy_{l+1}}) \to f_k(\overrightarrow{xy_{l+1}}) \langle C \rangle$,..., $f_i(\overrightarrow{xy_m}) \to f_k(\overrightarrow{xy_m}) \langle C \rangle$
$\} \cup R_2' \cup R_3$

For example, consider the following statement from the formula submission in Figure 3: **if** s isInstanceOf Imply **then** f(Not(s.l)) || f(s.r). Here, A_l corresponds to Imply, the set { A_1, ..., A_m } corresponds to the set of constructors of the type Formula, and the sequence $\overrightarrow{xy_1}$, ... , $\overrightarrow{xy_m}$ at this point in the translation corresponds to (s, True), (s, False), (s, Not(s_p)), (s, Imply(s_l, s_r)), resulting in the following rules:

```
f12(s, True) → f21(s, True) // fi → fk
f12(s, False) → f21(s, False) // fi → fk
f12(s, Not(s_p)) → f21(s, Not(s_p)) // fi → fk
f12(s, Imply(s_l, s_r)) → f13(s, Imply(s_l, s_r)) // fi → fi+1
```

where f13 and f21 lead to the **then** and **else** branches, respectively.

5 Evaluation

We next present our evaluation of termination checking techniques using our LCTRS export on existing benchmarks drawn from programming assignments. In our evaluation, we consider AProVE's Integer Term Rewrite Systems format (.itrs) [4] and use AProVE to prove their termination.[2]

We compare our results to two existing techniques in Stainless for proving termination: measure inference and measure transfer. Measure inference in Stainless was partially carried over from the Leon verifier [30], whose termination analysis was developed in the MSc thesis of Nicolas Voirol, along with support for generic types and quantifiers [31]. Subsequent work introduced a foundational type system that enforces termination [12]. Measure transfer was later introduced in [21], providing significant automation over measure inference for batched termination proving of equivalent programs, while requiring the original program to be annotated with a termination measure as a termination proof.

Table 1 shows the results of our evaluation. We consider the formula and sigma assignments from [20], as well as the prime and gcd assignments from the experience report in [19]. Each benchmark contains one or more reference solutions annotated with termination measures used for measure transfer, and equivalent student submissions with no measure annotations, where automated measure inference fails. Columns Inference and Transfer show the number of submissions successfully proven terminating by Stainless using measure inference and measure transfer, respectively. Column LCTRS shows number of submissions successfully proven terminating by AProVE.

In the formula benchmark, for 37 submissions where measure inference fails, the evaluation in [20] uses manual annotations to prove termination of 25 submissions (for the remaining submissions, the manual annotator did not find a termination measure). In contrast, measure transfer automatically proves correctness of 27 submissions (73%). The LCTRS export to AProVE proves termination of 24 submissions (65%), which is a significant improvement over measure inference in Stainless. Seven submissions could not be checked due to a limitation of our translation in handling Scala constructs. In the sigma benchmark, measure transfer succeeds for 678 submissions (96%), out of 704 submissions that required manual annotations in the previous evaluation in [20], due to limitations of measure inference. The 26 submissions where measure transfer fails are due to either equivalence proof failures (11) or due to introducing inner functions that exist only in the candidate submission (15). We were unable to evaluate our LCTRS export on this benchmark due to a lack of support for higher-order functions in AProVE's ITRSs. Our experiments show that, when removing the higher-order argument in the sigma submission, AProVE manages to prove termination, while measure inference in Stainless still fails. Since the

[2]Strictly speaking, ITRSs are not LCTRSs, but innermost termination of the LCTRSs when viewed as ITRSs implies innermost termination of the LCTRS, and our LCTRS export currently does not use features that are not supported by ITRSs.

Table 1: Evaluation results on programming assignments. Column LOC shows average number of lines of code per program. Columns F and D indicate average number of function definitions and average number of measure annotations per program, respectively. Columns R and S indicate the number of reference programs and number of submissions, respectively. The last three columns show the results of termination analysis. For each run, we set a 10 second timeout per Z3 solver query in Stainless and a 10 minute overall timeout for AProVE. Column Total Proven shows the total number of submissions successfully proven terminating, either using measure transfer or by AProVE.

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

higher-order argument does not affect termination in this case, in the future, we will explore program slicing techniques to remove higher-order arguments in the LCTRS export. In the `prime` benchmark, we encounter submissions that, when manually annotated, pass termination checks. However, measure transfer fails, because the inner function's measure in the reference solutions gives a negative measure when transferred to the inner function of the submission. LCTRS export outperforms measure transfer in this benchmark, succeeding for 64% of purely-functional submissions (compared to 23% when using measure transfer). Out of three benchmarks with multiple reference programs, only `gcd` has different termination measures. Termination analysis via the LCTRS export to AProVE outperforms measure inference in Stainless on this benchmark. However, this could be due to the use of unbounded integers in the LCTRS export, instead of bounded integers in the original submissions. When using unbounded integers in Stainless, measure inference succeeds for 31 submissions (out of 41 which were previously timing out due to bounded integer arithmetic).

Overall, our evaluation suggests that our approach using LCTRS export with AProVE provides a significant improvement over the measure inference in Stainless, but is still less effective than measure transfer. However, unlike our approach, measure transfer requires initial manual measure annotations in the reference program and is only useful when there is more than one program under analysis. Our approach using LCTRS export is more general.

6 Discussion and Future Work

Our work has several limitations in its current state.

Primitive Data Types So far, the only primitive data types that our exported LCTRSs contain are unbounded integers and booleans. The reason is that our current translation is backwards compatible to ITRSs as an early form of constrained rewriting [4]. This is why we currently export both unbounded and bounded Scala integers to unbounded integers on LCTRS level. Consequently, our termination proofs are correct only if there are no overflows in the original Scala program. To ensure correctness, Stainless checks for overflows prior to running the LCTRS export.

To illustrate the issues with the discrepancy between bounded integers in Scala and their translation as unbounded integers, we consider the following example program:

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

This program does not terminate due to a possible overflow of i for $n = 2^{31} - 1$. However, our translation would result in an LCTRS which uses unbounded integers and thus terminates. Currently, our use of the Scala-to-LCTRS translation is to prove termination of programs after having Stainless check for safety errors such as integer overflows and division by zero. In our overflow_fun example, Stainless will report addition overflow in this program, and the analysis pipeline never runs the termination check. Similarly, safety errors can occur in binary operations (Section 4.3). Stainless can detect such errors in an earlier phase prior to termination checks, omitting the need for LCTRS translation and termination checks.

LCTRSs provide more generality, both in theory and in practice: for example, not only do LCTRSs support bounded integers out of the box, but there are also dedicated termination proving techniques for such bounded integers [18]. A natural next step would be to export bounded Scala integers to bounded LCTRS integers.

Lazy Evaluation Our translation only supports a subset of Scala defined in Figure 1. Other advanced Scala features, such as **lazy val**s, are out of the scope of our work. In future work, we could adapt the approach used for proving termination of Haskell programs via term rewriting [8]. This approach uses a form of abstract interpretation [3] with the Haskell evaluation strategy to obtain rewrite rules whose *innermost* termination implies termination of the original Haskell program with the Haskell evaluation strategy. Alternatively/in addition, we could consider integrating lazy rewriting [26] into the target language of our translation, although this integration would require setting up more complex termination analysis infrastructure.

Higher-Order Functions Another limitation of the current state of our work is that our export does not support higher-order functions, a common feature in functional programs. This matters even for our benchmark set from a restricted application domain: recall that our translation was unable to handle the sigma benchmark. However, in the meantime LCTRSs have been extended with higher-order functions [10, 11] to a formalism called *Logically Constrained Simply-typed Term Rewriting Systems (LCSTRSs)*, motivated by translation-based termination analysis for functional programs (such as the present work). What is missing in the above work is support for techniques to simplify LCSTRSs with many rules. Very recent work [5] can help us overcome this limitation: in contrast to the earlier papers on LCSTRSs, this paper targets *innermost* and *call-by-value* termination rather than termination with regard to *arbitrary* rewrite strategies. This makes it possible to "chain" consecutive function calls, which makes systems much more amenable to automated termination analysis. As this addresses the main advantage of ITRSs, we plan to target LCSTRSs with Cora [16] as termination prover to benefit from their additional generality. Thus, an explicit removal of higher-order functions as part of the export may not be needed.

Modular Analysis A limitation of our export is that the LCTRS does not contain information which function symbols are allowed to occur in a start term of a possible infinite rewrite sequence. As a result, the termination prover analyzes termination for arbitrary start terms.

For Scala programs where a helper function would be non-terminating for some inputs on which it can never be called by the entry-point function, the resulting LCTRS would still be non-terminating from corresponding terms. This behavior is in line with current termination analysis in Stainless, which

```
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)
```

Figure 4: The main function invokes the non-terminating helper function under the condition x > y.

requires termination of all functions for all inputs.[3]

For example, the main function in Figure 4 invokes the helper function only when x > y holds. However, this information is not visible in the translation of the helper function. Because the helper function is not terminating when considered in isolation (due to the recursive call helper(x, y)), the resulting LCTRS is non-terminating.

Cora supports labeling symbols that should not be considered as possible start terms as "private" [10]. Adding such labels in our export would provide the needed information for a more powerful analysis.

Proof of Correctness Future work may include a mechanized proof of correctness of our translation, including formal semantics of the supported Scala subset and of LCTRS and a proof of preservation of termination and non-termination properties between the input and the generated output.

Propagation of information from the termination analysis to Scala Currently the only information from the termination analysis tool that reaches the user is whether termination was proved or disproved, or no output was found. This can be accompanied by a human-readable proof on rewrite level, but this has the downside that the user of Stainless cannot be expected to be familiar with the intricacies of termination analysis of term rewriting.

Future work will involve extracting information from the proof that can be presented to a Scala programmer. From a termination proof, a termination measure for a Scala function could be extracted and added as an annotation to the Scala function. From a non-termination proof, a non-terminating term could be extracted and translated back to Scala level. Here it would be necessary to check whether this non-terminating term would be reachable from a valid start location of the Scala function.

7 Conclusions

We have presented an approach for termination analysis of Scala programs via a translation to LCTRSs. Our approach allows for an automated extraction of an LCTRS such that innermost termination of the LCTRS implies termination of the Scala program. We have integrated our approach into the transformation pipeline in the Stainless program verifier. Our experiments on benchmarks from student programming assignments indicate complementarity to other approaches, such as the existing measure inference in Stainless or the measure transfer of manual measure annotations from related functions.

[3]In a system like Stainless, if the intention is to have a function that only expects certain inputs, then this should be reflected in the parameter data types and function preconditions.

References

[1] Martin Avanzini, Ugo Dal Lago & Georg Moser (2015): *Analysing the complexity of functional programs: higher-order meets first-order*. In: *Proc. ICFP*, pp. 152–164, doi:10.1145/2784731.2784753.

[2] Ştefan Ciobâcă, Dorel Lucanu & Andrei-Sebastian Buruiana (2023): *Operationally-based program equivalence proofs using LCTRSs*. *J. Log. Algebraic Methods Program.* 135, p. 100894, doi:10.1016/J.JLAMP.2023.100894.

[3] Patrick Cousot & Radhia Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Proc. POPL*, pp. 238–252, doi:10.1145/512950.512973.

[4] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp & Stephan Falke (2009): *Proving Termination of Integer Term Rewriting*. In: *Proc. RTA*, pp. 32–47, doi:10.1007/978-3-642-02348-4_3.

[5] Carsten Fuhs, Liye Guo & Cynthia Kop (2025): *An Innermost DP Framework for Constrained Higher-Order Rewriting*. In: *Proc. FSCD*, p. To appear.

[6] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Trans. Comput. Logic* 18(2), doi:10.1145/3060143.

[7] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski & René Thiemann (2017): *Analyzing Program Termination and Complexity Automatically with AProVE*. *J. Autom. Reason.* 58(1), pp. 3–31, doi:10.1007/S10817-016-9388-Y.

[8] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski & René Thiemann (2011): *Automated Termination Proofs for Haskell by Term Rewriting*. *ACM Trans. Program. Lang. Syst.*, doi:10.1145/1890028.1890030.

[9] Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes & Carsten Fuhs (2012): *Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs*. In: *Proc. PPDP*, pp. 1–12, doi:10.1145/2370776.2370778.

[10] Liye Guo, Kasper Hagens, Cynthia Kop & Deivid Vale (2024): *Higher-Order Constrained Dependency Pairs for (Universal) Computability*. In: *Proc. MFCS*, pp. 57:1–57:15, doi:10.4230/LIPICS.MFCS.2024.57.

[11] Liye Guo & Cynthia Kop (2024): *Higher-Order LCTRSs and Their Termination*. In: *Proc. ESOP (2)*, pp. 331–357, doi:10.1007/978-3-031-57267-8_13.

[12] Jad Hamza, Nicolas Voirol & Viktor Kunčak (2019): *System FR: Formalized Foundations for the Stainless Verifier*. *Proc. ACM Program. Lang.* 3(OOPSLA), pp. 166:1–166:30, doi:10.1145/3360592.

[13] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In: *Proc. FroCoS*, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[14] Cynthia Kop & Naoki Nishida (2015): *Constrained Term Rewriting tooL*. In: *Proc. LPAR*, pp. 549–557, doi:10.1007/978-3-662-48899-7_38.

[15] Cynthia Kop & Naoki Nishida (2015): *Converting C to LCTRSs*. https://www.trs.cm.is.nagoya-u.ac.jp/c2lctrs/formal.pdf.

[16] Cynthia Kop et al.: *The Cora Analyzer*. Available at `https://github.com/hezzel/cora`.

[17] EPFL LARA (2025): *Stainless*. `https://github.com/epfl-lara/stainless`.

[18] Ayuka Matsumi, Naoki Nishida, Misaki Kojima & Donghoon Shin (2023): *On Singleton Self-Loop Removal for Termination of LCTRSs with Bit-Vector Arithmetic*. *CoRR* abs/2307.14094, doi:10.48550/ARXIV.2307.14094. arXiv:2307.14094. In Proc. WST 2023.

[19] Dragana Milovančević, Mario Bucev, Marcin Wojnarowski, Samuel Chassot & Viktor Kunčak (2025): *Formal Autograding in a Classroom*. In: *Proc. ESOP*, pp. 154–174, doi:10.1007/978-3-031-91121-7_7.

[20] Dragana Milovančević & Viktor Kunčak (2023): *Proving and Disproving Equivalence of Functional Programming Assignments*. *Proc. ACM Program. Lang.* 7(PLDI), pp. 928–951, doi:10.1145/3591258.

[21] Dragana Milovančević, Carsten Fuhs, Mario Bucev & Viktor Kunčak (2024): *Proving Termination via Measure Transfer in Equivalence Checking*. In: *Proc. iFM*, pp. 75–84, doi:10.1007/978-3-031-76554-4_5.

[22] Georg Moser & Michael Schaper (2018): *From Jinja bytecode to term rewriting: A complexity reflecting transformation*. *Inf. Comput.* 261, pp. 116–143, doi:10.1016/J.IC.2018.05.007.

[23] Naoki Nishida & Misaki Kojima (2024): *CRaris: CR checker for LCTRSs in ARI Style*. In: *Proc. IWC*, pp. 83–84. In `http://cl-informatik.uibk.ac.at/iwc/iwc2024.pdf`.

[24] Martin Odersky, Lex Spoon & Bill Venners (2019): *Programming in Scala, Fourth Edition (A comprehensive step-by-step guide)*. Artima, Sunnyvale, CA, USA.

[25] Carsten Otto, Marc Brockschmidt, Christian Essen & Jürgen Giesl (2010): *Automated Termination Analysis of Java Bytecode by Term Rewriting*. In: *Proc. RTA*, pp. 259–276, doi:10.4230/LIPIcs.RTA.2010.259.

[26] Felix Schernhammer & Bernhard Gramlich (2007): *Termination of Lazy Rewriting Revisited*. In: *Proc. WRS*, pp. 35–51, doi:10.1016/J.ENTCS.2008.03.052.

[27] Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik & René Thiemann (2010): *Automated Termination Analysis for Logic Programs With Cut*. *Theory Pract. Log. Program.* 10(4-6), pp. 365–381, doi:10.1017/S1471068410000165.

[28] Jonas Schöpf & Aart Middeldorp (2025): *Automated Analysis of Logically Constrained Rewrite Systems using crest*. In: *Proc. TACAS*, pp. 124–144, doi:10.1007/978-3-031-90643-5_7.

[29] Dowon Song, Myungho Lee & Hakjoo Oh (2019): *Automatic and Scalable Detection of Logical Errors in Functional Programming Assignments*. *Proc. ACM Program. Lang.* 3(OOPSLA), pp. 188:1–188:30, doi:10.1145/3360614.

[30] Philippe Suter, Ali Sinan Köksal & Viktor Kuncak (2011): *Satisfiability Modulo Recursive Programs*. In: *Proc. SAS*, pp. 298–315, doi:10.1007/978-3-642-23702-7_23.

[31] Nicolas Voirol (2023): *Termination Analysis in a Higher-Order Functional Context*. Master's thesis, EPFL. Available at `http://infoscience.epfl.ch/record/311772`.

[32] Sarah Winkler & Georg Moser (2020): *Runtime Complexity Analysis of Logically Constrained Rewriting*. In: *Proc. LOPSTR*, pp. 37–55, doi:10.1007/978-3-030-68446-4_2.