Journal of Automated Reasoning (2023) 67:40
https://doi.org/10.1007/s10817-023-09669-z

Formal Verification of Termination Criteria for First-Order Recursive Functions

Cesar A. Muñoz[1] · **Mauricio Ayala-Rincón**[2] · **Mariano M. Moscato**[3] · **Aaron M. Dutle**[1] · **Anthony J. Narkawicz**[1] · **Ariane Alves Almeida**[4] · **Andréia B. Avelar da Silva**[5] · **Thiago M. Ferreira Ramos**[4]

Received: 10 September 2021 / Accepted: 24 April 2023 / Published online: 29 November 2023

Abstract
This paper presents a formalization of several termination criteria for first-order recursive functions. The formalization, which is developed in the Prototype Verification System (PVS), includes the specification and proof of equivalence of semantic termination, Turing termination, size change principle, calling context graphs, and matrix-weighted graphs. These termination criteria are defined on a computational model that consists of a basic functional language called PVS0, which is an embedding of recursive first-order functions. Through this embedding, the native mechanism for checking termination of recursive functions in PVS could be soundly extended with semi-automatic termination criteria such as calling contexts graphs.

Keywords Formal Verification · Termination · Calling Context Graph · PVS

1 Introduction

Advances in theorem proving have enabled the formal verification of algorithms used in safety-critical applications. For instance, the Prototype Verification System (PVS) [16] is

Cesar A. Muñoz, Aaron M. Dutle and Anthony J. Narkawicz have equally contributed to this work.

Work by Mariano M. Moscato funded by NASA/NIA Cooperative Agreement NNL09AA00A. Brazilian authors funded by CNPq Universal 409003/21-2, RG 313290/21-0 (M. Ayala-Rincón), and FAPDF DE 00193.00001175/21-11 grants.

✉ Mariano M. Moscato
mariano.m.moscato@nasa.gov

1 NASA Langley Research Center, Hampton, VA, USA

2 Departments of Computer Science and Mathematics, Universidade de Brasília, Brasília, Brazil

3 National Institute of Aerospace, Hampton, VA, USA

4 Department of Computer Science, Universidade de Brasília, Brasília, Brazil

5 Faculdade de Planaltina, Universidade de Brasília, Brasília, Brazil

extensively used at NASA in the verification of safety-critical algorithms of autonomous unmanned systems.[1] Many of these algorithms include the specification of recursive functions that, in PVS, are required to be complete, i.e., they are known to terminate for every possible input. In computer science, program termination is the quintessential example of a property that is undecidable. Alan Turing famously proved that it is impossible to construct a correct and complete algorithm that decides whether or not another algorithm terminates on a given input [22]. Turing's proof applies to algorithms written as Turing machines, but the proof extends to other formalisms for expressing computations, such as λ-calculus, rewriting systems, and programs written in modern programming languages.

As is the case for other undecidable problems, there are syntactic and semantic restrictions, data structures, and heuristics that lead to a solution for subclasses of the problem. In Coq, for example, termination of well-typed functions is guaranteed by the type system, a version of the Calculus of Inductive Constructions [5]. Other theorem provers, such as ACL2, have incorporated syntactic conditions for checking termination of recursive functions [10]. In PVS, users need to provide a measure function over a well-founded relation that strictly decreases at every recursive call [16]. Despite the undecidability result, showing termination for most specified functions is routine in computer science, but can often be a tedious and time-consuming stage in a formal verification effort.

This paper reports on the formalization of several termination criteria in PVS. In addition to the internal mechanism implemented in the type checker of PVS to assure termination of recursive definitions, this work also includes the formalization of more general techniques, such as the *size change principle* (SCP) presented by Lee et. al. [13]. The SCP principle states that if every infinite computation would give rise to an infinitely decreasing value sequence on a well-founded order, then no infinite computation is possible. Later, Manolios and Vroon introduced a particular concretization of the SCP, namely the Calling Context Graphs (CCG) and demonstrated its practical usefulness in the ACL2 prover [14]. Avelar's PhD dissertation proposes a refinement of the CCG technique, based on a particular algebra on matrices called Matrix-Weighted Graphs (MWG) [4]. The formalization reported in this paper includes all these criteria and proofs of equivalence between them. This paper also presents a practical contribution: a mechanizable technique to automate (some) termination proofs of user-defined recursive functions in PVS. While the formalization itself has been available for some time as part of the NASA PVS Library[2] and referenced in other works, e.g., [3] and [18], the main results were not published prior to the conference version [15] of this extended journal publication. Compared to the conference version, the present work includes expanded discussion, motivation, and examples in Sects. 2, 4, and 5; and a detailed description of Dutle's algorithm to check termination on matrix-weighted graphs, in Sect. 5.2. In addition to the previous elements, substantial extensions are given in Sect. 5.3 concerning automating the MWG technique in PVS, and in Sect. 5.4 that surveys recursive functions in the NASA PVS library,[3] with a discussion of how the MWG technique might apply.

For readability, this paper uses a stylized PVS notation. The development presented in this paper, including all lemmas, theorems, and examples, are formally verified in PVS and are available as part of the NASA PVS Library.

[1] For example, see https://shemesh.larc.nasa.gov/fm.

[2] https://github.com/nasa/pvslib/tree/master/PVS0 and https://github.com/nasa/pvslib/tree/master/CCG

[3] https://shemesh.larc.nasa.gov/fm/pvs.

2 PVS & PVS0

2.1 PVS

PVS is an interactive theorem prover based on classical higher-order logic. The PVS specification language is strongly-typed and supports several typing features including predicate sub-typing, dependent types, inductive data types, and parametric theories. The expressiveness of the PVS type system prevents its type-checking procedure from being decidable. Hence, the type-checker may generate proof obligations to be discharged by the user. These proof obligations are called *Type Correctness Conditions* (TCCs). The PVS system includes several pre-defined proof strategies that automatically discharge most of the TCCs.

In PVS, a recursive function f of type [A→B] is defined by providing a *measure function* M of type [A→T], where T is an arbitrary type, and a *well-founded relation* R over elements in T. The termination TCCs produced by PVS for a recursive function f guarantee that the measure function M strictly decreases with respect to R at every recursive call of f.

Example 1 The following PVS declaration defines a common version of the Ackermann function.

```
ackermann(m, n: ℕ) : RECURSIVE ℕ =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
  ELSE ackermann(m-1, ackermann(m,n-1))
  ENDIF
MEASURE lex2(m,n) BY <
```

In this example, the type A is the tuple $[\mathbb{N} \times \mathbb{N}]$ and the type B is $\mathbb{N}$. The type T is `ordinal`, the type denoting ordinal numbers in PVS. The measure function **lex2** maps a tuple of natural numbers into an ordinal number. Finally, the well-founded relation R is the order relation "<" on ordinal numbers. The termination-related TCCs generated by the PVS type-checker for the Ackermann function are shown in Fig. 1. Since all these TCCs are automatically discharged by a PVS built-in proof strategy, the PVS semantics guarantees that the function `ackermann` is well defined on all inputs.

For the Ackermann function above, *proving* termination is done automatically by existing built-in strategies in PVS. However, *determining* the measure function and in some cases, showing the well-foundedness of the relation can be more difficult. In this example, the lexicographic order on the inputs seems to be an obvious choice in hindsight, but it may take some careful inspection and thought to actually discover this is a useful measure function for this program. In addition, the function **lex2** was already specified in PVS, and the necessary order properties were proven so that the automated strategies would succeed. This is not always the case, and can slow the progress of specification and proof when it happens. One goal of the work described here is to enable this to be done automatically (cf. Sect. 5).

Note also that the termination TCCs generated here are created by the typechecking system of PVS. Because PVS does not have self-referential capabilities, reasoning about the correctness/completeness of alternative termination criteria cannot be performed directly inside the system itself. Thus, in order to do analysis of termination of programs, a (restricted) model of PVS was built inside of PVS. This model, called PVS0, is described below.

```
ackermann_TCC3: OBLIGATION
  ∀(m,n: ℕ): n = 0 ∧ m ≠ 0 ⇒ lex2(m-1, 1) < lex2(m, n)

ackermann_TCC5: OBLIGATION
  ∀(m,n: ℕ): n ≠ 0 ∧ m ≠ 0 ⇒ lex2(m, n-1) < lex2(m, n)

ackermann_TCC6: OBLIGATION
  ∀(m,n: ℕ, f: [{z: [ℕ×ℕ] | lex2(z`1, z`2) < lex2(m,n)} → ℕ]):
    n ≠ 0 ∧ m ≠ 0 ⇒ lex2(m-1, f(m, n-1)) < lex2(m,n)
```

Fig. 1 Termination-related TCCs for the Ackermann function in Ex. 1

2.2 PVS0

2.2.1 PVS0 Language Definition

PVS0 is a basic functional language used in this paper as a computational model for first-order recursive functions in PVS. More precisely, PVS0 is an embedding of univariate first-order recursive functions of type $[\mathscr{V}\to\mathscr{V}]$ for an arbitrary generic type $\mathscr{V}$. The syntactic expressions of PVS0 are defined by the grammar

$$e ::= \texttt{cnst}(v) \mid \texttt{vr} \mid \texttt{op1}(n, e) \mid \texttt{op2}(n, e, e) \mid \texttt{rec}(e) \mid \texttt{ite}(e, e, e),$$

where v is a value of type $\mathscr{V}$ and n is a natural number. Furthermore, $\texttt{cnst}(v)$ denotes a constant with value v, vr denotes the unique variable in the language, op1 and op2 denote unary and binary operators respectively, rec denotes a recursive call, and ite denotes a conditional expression ("if-then-else"). The first parameter of op1 and op2 is an index used to identify built-in operators of type $[\mathscr{V}\to\mathscr{V}]$ and $[[\mathscr{V}\times\mathscr{V}]\to\mathscr{V}]$, respectively. In the following, the collection of PVS0 expressions is referred to as $\mathscr{E}_{\mathscr{V}}$, where the type parameter for $\mathscr{E}$ is omitted when possible to lighten the notation. The PVS0 programs with values in $\mathscr{V}$, denoted by $\texttt{PVS0}_{\mathscr{V}}$, are 4-tuples of the form $(O_1, O_2, \perp, e)$, such that

- O_1 is a list of unary operators of type $[\mathscr{V}\to\mathscr{V}]$, where $O_1(i)$, i.e., the i-th element of the list O_1, interprets the index i as referred by in the application of op1,
- O_2 is a list of binary operators of type $[[\mathscr{V}\times\mathscr{V}]\to\mathscr{V}]$, where $O_2(i)$ interprets the index i in applications of op2,
- $\perp$ is a constant of type $\mathscr{V}$ representing the Boolean value *false* in the conditional construction ite, and
- e is an expression from $\mathscr{E}$: the syntactic representation of the body of the program.

The operators in O_1 and O_2 are PVS pre-defined functions, whose evaluation is considered to be atomic in the proposed computational model. These operators make it easy to modularly embed first-order PVS recursive functions in PVS0, while maintaining non-recursive PVS functions directly available to PVS0 definitions. Henceforth, if $\texttt{p} = (O_1, O_2, \perp, e)$ is a PVS0 program, the symbols $\texttt{p}_{O_1}$, $\texttt{p}_{O_2}$, $\texttt{p}_{\perp}$, and $\texttt{p}_e$ denote, respectively, the first, second, third, and fourth elements of the tuple.

Since there is only one variable available to write PVS0 programs, arguments of binary functions such as Ackermann's need to be encoded in $\mathscr{V}$, for example using tuples as shown in Example 2.

Example 2 The Ackermann function of Example 1 can be implemented as the $\texttt{PVS0}_{[\mathbb{N}\times\mathbb{N}]}$ program $\texttt{ack} \equiv (O_1, O_2, \perp, e)$, where the type parameter $\mathscr{V}$ of PVS0 is instantiated with the type of pairs of natural numbers, i.e., $[\mathbb{N}\times\mathbb{N}]$. In this encoding, the first projection of the result of the program represents the output of the function. The components of `ack` are defined below.

- $O_1(0)((m,n)) \equiv$ `IF` $m = 0$ `THEN` $\top$ `ELSE` $\bot$ `ENDIF` .
- $O_1(1)((m,n)) \equiv$ `IF` $n = 0$ `THEN` $\top$ `ELSE` $\bot$ `ENDIF` .
- $O_1(2)((m,n)) \equiv (n+1, 0)$.
- $O_1(3)((m,n)) \equiv$ `IF` $m = 0$ `THEN` $\bot$ `ELSE` $(\max(0, m-1), 1)$ `ENDIF` .
- $O_1(4)((m,n)) \equiv$ `IF` $n = 0$ `THEN` $\bot$ `ELSE` $(m, \max(0, n-1))$ `ENDIF` .
- $O_2(0)((m,n),(i,j)) \equiv$ `IF` $m = 0$ `THEN` $\bot$ `ELSE` $(\max(0, m-1), i)$ `ENDIF` .
- $\bot \equiv (0,0)$, and for convenience $\top \equiv (1,0)$.
- e $\equiv$
```
ite(op1(0,vr), op1(2,vr),
    ite(op1(1,vr), rec(op1(3,vr)),
        rec(op2(0,vr,rec(op1(4,vr)))))).
```

Example 2 illustrates the use of built-in operators in PVS0. Any unary or binary PVS function can be used as an operator in the construction of a PVS0 program. In order to show that `ack` correctly encodes the Ackermann function, it is necessary to define the operational semantics of PVS0.

2.2.2 Semantic Relation

Given a PVS0 program p, the semantic evaluation of an $\mathscr{E}$ expression e_i is given by the relation ε defined as follows. Intuitively, it holds when given a subexpression e_i of a program p, the evaluation of e_i on the input value v_i results in the output value v_o.

Definition 1 (*Semantic Relation*) Let p be a PVS0 program on a generic type $\mathscr{V}$, e_i be an expression in $\mathscr{E}$, and v_i, v_o, v, v', v'' be values from $\mathscr{V}$. The relation $\varepsilon(\mathtt{p})(e_i, v_i, v_o)$ holds if and only if

$$\begin{cases} v_o = v & \text{if } e_i = \mathtt{cnst}(v) \\ v_o = v_i & \text{if } e_i = \mathtt{vr} \\ \exists\, v' : \varepsilon(\mathtt{p})(e_1, v_i, v') \wedge v_o = \chi_1(\mathtt{p})(j, v') & \text{if } e_i = \mathtt{op1}(j, e_1) \\ \exists\, v', v'' : \varepsilon(\mathtt{p})(e_1, v_i, v') \wedge \varepsilon(\mathtt{p})(e_2, v_i, v'') & \\ \qquad \wedge\, v_o = \chi_2(\mathtt{p})(j, v', v'') & \text{if } e_i = \mathtt{op2}(j, e_1, e_2) \\ \exists\, v' : \varepsilon(\mathtt{p})(e_1, v_i, v') \wedge \varepsilon(\mathtt{p})(\mathtt{p}_e, v', v_o) & \text{if } e_i = \mathtt{rec}(e_1) \\ \exists\, v' : \varepsilon(\mathtt{p})(e_1, v_i, v') \wedge (v' \neq \mathtt{p}_\bot \Rightarrow \varepsilon(\mathtt{p})(e_2, v_i, v_o)) & \\ \qquad \wedge\, (v' = \mathtt{p}_\bot \Rightarrow \varepsilon(\mathtt{p})(e_3, v_i, v_o)) & \text{if } e_i = \mathtt{ite}(e_1, e_2, e_3) \end{cases}$$

where

$$\chi_1(\mathtt{p})(j, v) = \begin{cases} \mathtt{p}_{O_1}(j)(v) & \text{if } j < |\mathtt{p}_{O_1}| \\ \mathtt{p}_\bot & \text{otherwise.} \end{cases}$$

$$\chi_2(\mathtt{p})(j, v_1, v_2) = \begin{cases} \mathtt{p}_{O_2}(j)(v_1, v_2) & \text{if } j < |\mathtt{p}_{O_2}| \\ \mathtt{p}_\bot & \text{otherwise.} \end{cases}$$

The following lemma states that the `ack` program encodes the function `ackermann`.

Lemma 1 *For all $n, m, k \in \mathbb{N}$, $\mathtt{ackermann}(m,n) = k$ if and only if there exists $i \in \mathbb{N}$ such that $\varepsilon(\mathtt{ack})(\mathtt{ack}_e, (m,n), (k,i))$.*

The proof of Lemma 1 proceeds by structural induction on the definition of the function `ackermann` and the relation ε. A proof of this kind of statement is usually tedious and long. However, it is fully mechanizable in PVS assuming that the function and the PVS0 program share the same syntactical structure. A proof strategy that automatically discharges equivalences between PVS functions and PVS0 programs was developed.

The following theorem shows that the semantic relation ε is deterministic.

Theorem 1 *Let* $\mathtt{p}$ *be a* PVS0 *program. For any expression* $e_i \in \mathscr{E}$ *and any pair of values* $v_i, v'_o, v''_o \in \mathscr{V}$,

$$\varepsilon(\mathtt{p})(e_i, v_i, v'_o) \text{ and } \varepsilon(\mathtt{p})(e_i, v_i, v''_o) \text{ implies } v'_o = v''_o.$$

PVS0 enables the encoding of non-terminating functions. The predicate ε-*determined*, defined below, holds when a PVS0 program encodes a function that returns a value for a given input.

Definition 2 (ε-*determination*) A `PVS0` program $\mathtt{p}$ is said to be ε-*determined* for an input value $v_i \in \mathscr{V}$ (denoted by $D_\varepsilon(\mathtt{p}, v_i)$) when $\exists v_o \in \mathscr{V} : \varepsilon(\mathtt{p})(\mathtt{p}_e, v_i, v_o)$.

2.2.3 Functional Semantics

The operational semantics of PVS0 can be expressed by a function $\chi : [\mathtt{PVS0} \to [\mathscr{E} \times \mathscr{V} \times \mathbb{N}] \to \mathscr{V} \uplus \{\diamond\}]$. This function returns either a value of type $\mathscr{V}$ or a distinguished value $\diamond \notin \mathscr{V}$. The natural number argument represents an upper bound on the number of nested recursive calls that are to be evaluated. If this bound is reached and no final value has been computed, the function returns $\diamond$.

Definition 3 (*Semantic Function*) Let $\mathtt{p}$ be a `PVS0` program, e_i an expression from $\mathscr{E}$, v_i a value from $\mathscr{V}$, n a natural number, $v' = \chi(\mathtt{p})(e_1, v_i, n)$, and $v'' = \chi(\mathtt{p})(e_2, v_i, n)$.

$$\chi(\mathtt{p})(e_i, v_i, n) \equiv \begin{cases} v & \text{if } n > 0 \text{ and } e_i = \mathtt{cnst}(v) \\ v_i & \text{if } n > 0 \text{ and } e_i = \mathtt{vr} \\ \chi_1(\mathtt{p})(j, v') & \text{if } n > 0, e_i = \mathtt{op1}(j, e_1), \text{ and } v' \neq \diamond \\ \chi_2(\mathtt{p})(j, v', v'') & \text{if } n > 0, e_i = \mathtt{op2}(j, e_1, e_2), \\ & \ v' \neq \diamond, \text{ and } v'' \neq \diamond \\ \chi(\mathtt{p})(e, v', n-1) & \text{if } n > 0, e_i = \mathtt{rec}(e_1), \text{ and } v' \neq \diamond \\ \chi(\mathtt{p})(e_2, v_i, n) & \text{if } n > 0, e_i = \mathtt{ite}(e_1, e_2, e_3), v' \neq \diamond, \text{ and } v' \neq \mathtt{p}_\perp \\ \chi(\mathtt{p})(e_3, v_i, n) & \text{if } n > 0, e_i = \mathtt{ite}(e_1, e_2, e_3), v' \neq \diamond, \text{ and } v' = \mathtt{p}_\perp \\ \diamond & \text{otherwise.} \end{cases}$$

The well-definedness of the function χ is proved using the well-founded measure given by the lexicographical order of the pairs n measured as a natural number, and the size of the subexpression e_1 being evaluated by the semantic function. As discussed in the begining of the section on PVS & `PVS0`, all recursive functions are specified jointly with a well-founded measure provided by the specifier. In addition, PVS generates termination proof obligations (TCCs) that should be discharged either automatically or, when not possible, manually by the specifier.

The following theorem states that the semantic relation ε and the semantic function χ are equivalent.

Theorem 2 *For any* PVS0 *program* $\mathtt{p}$, $v_i, v_o \in \mathscr{V}$ *and* $e_i \in \mathscr{E}$, $\varepsilon(\mathtt{p})(e_i, v_i, v_o)$ *if and only if* $v_o = \chi(\mathtt{p})(e_i, v_i, n)$, *for some* $n \in \mathbb{N}$.

A program $\mathtt{p}$ is χ-determined for an input v_i, as defined below, if the evaluation of $\mathtt{p}(v_i)$ produces a value in a finite number of nested recursive calls.

Definition 4 (χ*-determination*) A $\mathtt{PVS0}$ program $\mathtt{p}$ is said to be χ*-determined* for an input value $v_i \in \mathscr{V}$ (denoted by $D_\chi(\mathtt{p}, v_i)$) when there is an $n \in \mathbb{N}$ such that $\chi(\mathtt{p})(\mathtt{p}_e, v_i, n) \neq \diamondsuit$.

The equivalence of ε-determination and χ-determination is a corollary of Theorem 2.

Theorem 3 *For all* $\mathtt{p} \in \mathtt{PVS0}_{\mathscr{V}}$ *and value* $v_i : \mathscr{V}$, $D_\varepsilon(\mathtt{p}, v_i)$ *if and only if* $D_\chi(\mathtt{p}, v_i)$.

In Definition 4, there may be multiple (in fact, infinite) natural numbers n that satisfy $\chi(\mathtt{p})(\mathtt{p}_e, v_i, n) \neq \diamondsuit$. The following definition distinguishes the minimum of those numbers.

Definition 5 (μ) Let $\mathtt{p}$ be a PVS0 program and v_i a value in $\mathscr{V}$ such that $D_\chi(\mathtt{p}, v_i)$, the minimum number of recursive calls needed to produce a result (denoted by $\mu(\mathtt{p}, v_i)$) is formally defined as $\min(\{n \in \mathbb{N} \mid \chi(\mathtt{p})(\mathtt{p}_e, v_i, n) \neq \diamondsuit\})$.

If $\mathtt{p}$ is χ-determined for a value v_i, then for any $n \geq \mu(\mathtt{p}, v_i)$ the evaluation of $\chi(\mathtt{p})(\mathtt{p}_e, v_i, n)$ results in a value from $\mathscr{V}$. This remark is formalized by the following lemma.

Lemma 2 *Let* $\mathtt{p}$ *be a* $\mathtt{PVS0}$ *program and* v_i *a value from* $\mathscr{V}$ *such that* $D_\chi(\mathtt{p}, v_i)$. *For any* $n \in \mathbb{N}$ *such that* $n \geq \mu(\mathtt{p}, v_i)$, $\chi(\mathtt{p})(\mathtt{p}_e, v_i, n) = \chi(\mathtt{p})(\mathtt{p}_e, v_i, \mu(\mathtt{p}, v_i))$.

2.2.4 Semantic Termination

The notion of termination for PVS0 programs is defined using the notions of determination from Sect. 2.2.3.

Definition 6 (ε*-termination* and χ*-termination*) A $\mathtt{PVS0}$ program $\mathtt{p} \in \mathtt{PVS0}_{\mathscr{V}}$ is said to be ε*-terminating* (noted $T_\varepsilon(\mathtt{p})$) when $\forall v_i \in \mathscr{V} : D_\varepsilon(\mathtt{p}, v_i)$. It is said to be χ*-terminating* ($T_\chi(\mathtt{p})$) when $\forall v_i \in \mathscr{V} : D_\chi(\mathtt{p}, v_i)$.

As a corollary of Theorem 3, the notions of ε-termination and χ-termination coincide.

Theorem 4 *For every* $\mathtt{PVS0}$ *program* $\mathtt{p}$, $T_\varepsilon(\mathtt{p})$ *if and only if* $T_\chi(\mathtt{p})$.

Not all PVS0 programs are terminating. For example, consider the PVS0 program $\mathtt{p}'$ with body $\mathtt{rec}(\mathtt{vr})$. It can be proven that $D_\varepsilon(\mathtt{p}', v_i)$ does not hold for any $v_i \in \mathscr{V}$. Hence, $T_\varepsilon(\mathtt{p}')$ does not hold and, equivalently, nor does $T_\chi(\mathtt{p}')$. Since terminating programs compute a value for every input, the function χ can be refined into an evaluation function for terminating programs that does not depend on the existence of a distinguished value outside $\mathscr{V}$, such as $\diamondsuit$.

Definition 7 Let $\mathtt{PVS0}_{\downarrow_\varepsilon}$ be the collection of PVS0 programs for which T_ε holds, let $\mathtt{p} \in \mathtt{PVS0}_{\downarrow_\varepsilon}$, and v_i be a value in $\mathscr{V}$. The semantic function for terminating programs $\epsilon : [\mathtt{PVS0}_{\downarrow_\varepsilon} \to \mathscr{V} \to \mathscr{V}]$ is defined in the following way.

$\epsilon(\mathtt{p})(v_i) \equiv \epsilon_e(\mathtt{p})(\mathtt{p}_e, v_i)$, where $v' = \epsilon_e(\mathtt{p})(e_1, v_i)$, $v'' = \epsilon_e(\mathtt{p})(e_2, v_i)$, and

$$\epsilon_e(\mathtt{p})(e_i, v_i) \equiv \begin{cases} v & \text{if } e_i = \mathtt{cnst}(v) \\ v_i & \text{if } e_i = \mathtt{vr} \\ \chi_1(\mathtt{p})(j,\ v') & \text{if } e_i = \mathtt{op1}(j, e_1) \\ \chi_2(\mathtt{p})(j,\ v',\ v'') & \text{if } e_i = \mathtt{op2}(j, e_1, e_2) \\ \epsilon_e(\mathtt{p})(e, v') & \text{if } e_i = \mathtt{rec}(e_1) \\ \epsilon_e(\mathtt{p})(e_2, v_i) & \text{if } e_i = \mathtt{ite}(e_1, e_2, e_3) \text{ and } \epsilon_e(\mathtt{p})(e_1, v_i) \neq \mathtt{p}_\perp \\ \epsilon_e(\mathtt{p})(e_3, v_i) & \text{if } e_i = \mathtt{ite}(e_1, e_2, e_3) \text{ and } \epsilon_e(\mathtt{p})(e_1, v_i) = \mathtt{p}_\perp \end{cases}$$

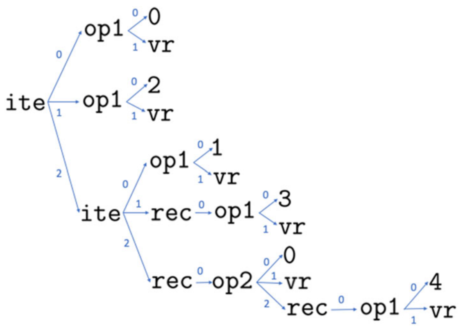


Fig. 2 Abstract syntax tree of the Ackermann function from Example 2

Notice that the definition above holds only for ε-terminating programs. Therefore, a well-founded ordering exists to measure the needed number of nested recursive calls to evaluate any subexpression e_i of the PVS0 program p with input v_i. The specification of this function comes with the measure function given by the lexicographical order of pairs given by the number of necessary nested recursive calls measured as natural numbers and the size of PVS0 expressions. Using this lexicographic measure, it is possible to manually prove the well-definedness of this function built by PVS as a termination TCC that the system cannot prove automatically.

Theorem 5 *For all terminating PVS0 program* $\mathtt{p}$*, i.e.,* $T_\varepsilon(\mathtt{p})$ *holds, and values* $v_i, v_o \in \mathscr{V}$*,* $\varepsilon(\mathtt{p})(\mathtt{p}_e, v_i, v_o)$ *holds if and only if* $\epsilon(\mathtt{p})(v_i) = v_o$*.*

While T_ε and T_χ provide semantic definitions of termination, these definitions are impractical as termination criteria, since they involve an exhaustive examination of the whole universe of values in $\mathscr{V}$. The rest of this paper formalizes termination criteria that yield mechanical termination analysis techniques.

3 Turing Termination Criterion

In contrast to the purely semantic notions of termination presented in Sect. 2, the so-called *Turing termination criterion* relies on the syntactic structure of recursive programs. In particular, this termination criterion uses a characterization of the input values that lead to the evaluation of recursive call subexpressions, i.e., rec(e). In order to define such a characterization, it is necessary to formalize a way to identify univocally a particular subexpression of a given PVS0 program. Furthermore, the subexpression as well as its actual position must be identified. If a given program body contains several repetitions of the same expression, such as op2(0,rec(vr),rec(vr)), which has two occurrences of rec(vr), the criterion needs them to be distinguishable from one another. Such a reference for subexpressions can be formally defined using the abstract syntax tree of the enclosing expression. To illustrate the idea, Fig. 2 depicts a graphical representation of the abstract syntax tree of the ack program. A unique identifier for a given subexpression can be constructed by collecting all the numbers labeling the edges from the subexpression to the root of the tree. For example, the sequence of numbers that identify the subexpression rec(op1(4,vr)) is $\langle 2, 0, 2, 2\rangle$. A syntax tree labeled using these sequences is called a *labeled syntax tree*.

Definition 8 (*Valid Path*) Let p be a PVS0 program, a finite sequence of natural numbers p is a *Valid Path* of p if p determines a path in the labeled syntax tree of p from any node e to the root of the tree. In that case, p is said to *reach* e in p.

The notion of path is strictly syntactic. Nevertheless, a semantic correlation is also needed to state termination criteria focused on how the inputs change along successive recursive calls, as is the case for Turing termination criterion. A semantic way to identify a subexpression e of a given program p is to recognize all the values that reach the particular subexpression e when used as inputs for the evaluation of p. Here, a value whose execution induces a path which contains e is said to reach the subexpression. It is possible to characterize such values by collecting all the expressions that act as guards for the conditional expressions traversed for a given path reaching e.

Continuing the example based on the ack program, for the path $\langle 2, 0, 2, 2\rangle$ reaching `rec(op1(4,vr))`, such expressions would be `op1(0,vr)` and `op1(1,vr)`. For that specific path, the values to be characterized are the ones that falsify both guard expressions, i.e., the values for which both expressions evaluate to $\mathtt{p}_\perp$. Nevertheless, for the path $\langle 1, 2\rangle$ reaching `rec(op1(3,vr))`, the collected expressions are the same, but it is necessary for the latter not to evaluate to $\mathtt{p}_\perp$ in order to characterize the input values that would exercise `rec(op1(3,vr))`.

The previous example shows that it is necessary not only to collect the guard expressions, but also to determine whether each one needs to evaluate to $\mathtt{p}_\perp$ or not.

Definition 9 (*Polarized Expression*) Given an expression $e \in \mathscr{E}$, the *polarized version of* e is a pair $[\mathscr{E} \times \{0, 1\}]$ such that $(e, 0)$, abbreviated as $\neg e$, indicates that e should evaluate to $\mathtt{p}_\perp$ and the pair $(e, 1)$, which is abbreviated simply as e, indicates the contrary.

For a given program p, an input value v_i, and a polarized expression $c = (e, b)$ with $b \in \{0, 1\}$, c is said to be *valid* when the condition expressed by it holds. The predicate $\varepsilon_\pm$ defined below formalizes this notion.

$$\varepsilon_\pm(\mathtt{p})(c, v_i) \equiv \begin{cases} \varepsilon(\mathtt{p})(e, v_i, \mathtt{p}_\perp) & \text{if } b = 0, \\ \neg\varepsilon(\mathtt{p})(e, v_i, \mathtt{p}_\perp) & \text{otherwise.} \end{cases}$$

The semantic characterization of a particular subexpression is formalized by the notion of list of path conditions defined below.

Definition 10 (*Path Conditions*) Let p be a valid path of a PVS0 program p and e the subexpression of $\mathtt{p}_e$ reached by p. The list of polarized guard expressions of p that are needed to be valid in order for the evaluation of p to involve the expression e is called the *list of path conditions* of p.

Definition 11 (*Calling Context*) A *calling context* of a program p is a tuple $(\mathtt{rec}(e'), p, \mathbf{c})$ containing: a path p, which is valid in p, a recursive-call expression $\mathtt{rec}(e')$ contained in $\mathtt{p}_e$ and reached by p, and the list $\mathbf{c}$ of path conditions of p. The collection of all calling contexts of p is denoted by $\mathbf{cc}(\mathtt{p})$.

The notion of calling context captures both the syntactic and the semantic characterizations of the subexpressions of a program that denote recursive calls.

Example 3 The calling contexts for the ack function from Example 2 are:

– (`rec(op1(3,vr))`, $\langle 1, 2\rangle$, $\langle\neg$`op1(0,vr)`, `op1(1,vr)`$\rangle$),
– (`rec(op2(0,vr,rec(op1(4,vr))))`, $\langle 2, 2\rangle$, $\langle\neg$`op1(0,vr)`, $\neg$`op1(1,vr)`$\rangle$), and

– $(\texttt{rec(op1(4,vr))}, \langle 2,0,2,2\rangle, \langle \neg\texttt{op1(0,vr)}, \neg\texttt{op1(1,vr)}\rangle)$.

A value whose execution leaves a `PVS0` program to evaluate the expression e, argument of a recursive call in a calling context $(e, p, \mathbf{c})$, of a `PVS0` program is said to exercise the calling context as defined below.

Definition 12 (*Exercising values*) Given a `PVS0` program p, an input value v is said to *exercise* a calling context $\mathbf{cc} = (e, p, \mathbf{c})$ in p when $\varepsilon_{\pm}(\mathrm{p})(c, v)$ holds for all $c \in \mathbf{c}$.

A program p is TCC-terminating if for each calling context $\mathbf{cc}$ in p and every input value v_i exercising $\mathbf{cc}$, the value of the expression used as argument by the call in $\mathbf{cc}$ is smaller than v_i. In this context, a value v_1 is considered smaller than a value v_2 if, considering the entire order, the shortest path to some minimal element is shorter for v_1 than for v_2.

Definition 13 (*TCC-termination*) A PVS0 program p is said to be *TCC-terminating*, or *Turing-terminating*, on a measuring type M if there exist a function $m : [\mathscr{V} \to M]$ and a well-founded relation $<_M$ on M such that for all calling context $\mathbf{cc} = (\texttt{rec}(e), p, \mathbf{c})$ among the calling contexts of p, for all $v_i, v_o \in \mathscr{V}$, if $\varepsilon_{\pm}(\mathrm{p})(\mathbf{c}, v_i)$ and $\varepsilon(\mathrm{p})(e, v_i, v_o)$ hold, then $m(v_o) <_M m(v_i)$.

The notion of TCC-termination on a program p is denoted by the predicate $T_T^{[M,<_M,m]}(\mathrm{p})$, which is parametric on the measure type M, the well-founded relation $<_M$, and the measure function m. TCC-termination is equivalent to ε-termination (and, therefore, to χ-termination) as stated by Theorem 6 below. A key construction used in the proof of Theorem 6 is the function Ω, defined as follows.

Definition 14 (Ω) Let $<_{\mathrm{p},m}$ be a binary relation on $\mathscr{V}$ defined as $v_1 <_{\mathrm{p},m} v_2$ if and only if $m(v_1) <_M m(v_2)$ and the value v_2 exercises the calling context $\mathbf{cc} = (e, p, \mathbf{c})$ in the program p such that $\varepsilon(\mathrm{p})(e, v_2, v_1)$ holds.

Then, $\Omega_{\mathrm{p},m}(v) \equiv \min(\{i : \mathbb{N}^+ \mid \forall\, v' \in \mathscr{V} : \neg(v' <^i_{\mathrm{p},m} v)\})$

where $v' <^i_{\mathrm{p},m} v$ denotes a chain of $i+1$ values related by $<_{\mathrm{p},m}$ with endpoints in v' and v.

Example 4 The function Ω for the Ackermann program `ack` in Example 2 is given as below.

$$\begin{array}{lcl}
\Omega((0,n)) & = & 1 \\
\Omega((1,n)) & = & n+2 \\
\Omega((m+2,0)) & = & 1+\Omega((m+1,1)) \\
\Omega((m+2,n+1)) & = & 1+\Omega((m+1,\texttt{ackerman}(m+2,n)))
\end{array}$$

Thus, $\Omega((2,0)) = 4$, and $\Omega((2,n+1)) = 2(n+1)+4$; $\Omega((3,0)) = 9$, and $\Omega((3,n+1)) = 1 + 2^{n+3}$; $\Omega((4,0)) = 18$, and $\Omega((4,n+1)) = 2 + \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{(n+1)+3 \text{ times}}$; etc.

The following lemma states that the function Ω acts as an upper bound for semantic termination.

Lemma 3 *Let p be a TCC-terminating `PVS0` program, i.e., p satisfies $T_T^{[M,<_M,m]}(p)$ for a measure type M, a well-founded relation $<_M$ over M, and a measure function m. For any values $v_i, v_o \in \mathscr{V}$, and $e \in \mathscr{E}$, $\varepsilon(p)(e, v_i, v_o)$ implies $v_o = \chi(p)(e, v_i, n)$, for some $n \in \mathbb{N}$, such that $n \leq \Omega_{p,m}(v_i)$.*

Proof The lemma is formalized considering the evaluation of expressions at valid paths of the program $\mathtt{p}$ and path conditions. Essentially, the more general property below is proved.

For any valid path p, and expression e of $\mathtt{p}$ reached by p, such that the associated path conditions hold for input value v, there exists $n \leq \Omega_{\mathtt{p},m}(v)$, such that $\chi(\mathtt{p})(e, v, n) \neq \diamond$ and $\varepsilon(\mathtt{p})(e, v, \chi(\mathtt{p})(e, v, n))$. The proof requires induction on pairs of environments and expressions, using the lexicographic order built from the well-founded relation $<_M$ on values and the size of expressions. The tricky part of the proof is that it captures more granularity than TCC-termination since it states the decrement of the measure through all traces of the pprogram $\mathtt{p}$ during evaluation. Indeed, the valid path p and the associated expression e do not necessarily correspond to a path to a recursive call and the subexpression of the recursive call. □

The following lemma states a relation between μ, the number of nested recursive calls in the evaluation of a particular input v, and $\Omega_{\mathtt{p},m}$ for the same input v.

Lemma 4 *Let $\mathtt{p}$ be a TCC-terminating $\mathtt{PVS0}$ program, i.e., $\mathtt{p}$ satisfies $T_T^{[M,<_M,m]}(\mathtt{p})$ for a measure type M, a well-founded relation $<_M$ over M, and a measure function m. For any value $v \in \mathscr{V}$, $\mu(\mathtt{p}, v) \leq \Omega_{\mathtt{p},m}(v)$.*

Proof The proof proceeds by showing that $\Omega_{\mathtt{p},m}(v) \in \{n \in \mathbb{N} \mid \chi(\mathtt{p})(e, v, n) \neq \diamond\}$, i.e., proving that $\Omega_{\mathtt{p},m}(v)$ is an upper bound of the minimum of this set, $\mu(\mathtt{p}, v)$. The fact that such set is nonempty is a consequence of the assumption that $\mathtt{p}$ is a TCC-terminating $\mathtt{PVS0}$ program. □

Theorem 6 *Let $\mathtt{p}$ be a PVS0 program, $T_\varepsilon(\mathtt{p})$ holds if and only if there exist a measure type M, a well-founded relation $<_M$ on M, and a measure function m such that $T_T^{[M,<_M,m]}(\mathtt{p})$ holds as well.*

Proof Assuming $T_\varepsilon(\mathtt{p})$, it can be proved that $T_T^{[\mathbb{N},<,\mu_{\mathtt{p}}]}(\mathtt{p})$ holds, where $\mu_{\mathtt{p}}(v) = \mu(\mathtt{p}, v)$. The function $\mu_{\mathtt{p}}(v)$ is well defined for every v since $T_\varepsilon(\mathtt{p})$ holds and then, by Theorem 4, $D_\chi(\mathtt{p}, v)$ holds as well. Following the definition of χ and the determinism of ε (Lemma 1), it can be seen that $\mu_{\mathtt{p}}(v_o) < \mu_{\mathtt{p}}(v_i)$ for each pair of values v_i, v_o such that $\varepsilon_\pm(\mathtt{p})(\mathbf{c}, v_i)$ and $\varepsilon(\mathtt{p})(e, v_i, v_o)$ for every calling context $(\mathtt{rec}(e), p, \mathbf{c})$ in $\mathtt{p}$. The opposite implication can be proved stating that if $T_T^{[M,<_M,m]}(\mathtt{p})$ holds, for every $v \in \mathscr{V}$ and any subexpression e of $\mathtt{p}$, there exists a natural number $n \leq \Omega_{\mathtt{p},m}(v)$ such that $\chi(\mathtt{p})(e, v_i, n) \neq \diamond$, which assures $T_\varepsilon(\mathtt{p})$ by Theorem 4. The proof of such a property proceeds by induction on the lexicographic order given by $(m(v), |e|)$, where $|e|$ denotes the size of the expression e. □

Theorem 6 can be used as a practical tool to prove ε-termination of $\mathtt{PVS0}$ programs, as illustrated by the following lemma.

Lemma 5 *The $\mathtt{PVS0}$ program $\mathtt{ack}$ from Example 2 is ε-terminating, i.e., $T_\varepsilon(\mathtt{ack})$ holds.*

Proof In order to use the Theorem 6, it is necessary to prove first that there exist a measure type M, a well-founded relation $<_M$ over M, and a measure function m such that $T_T^{[M,<_M,m]}(\mathtt{ack})$ holds. Let M be the type of pairs of natural numbers $[\mathbb{N} \times \mathbb{N}]$, m the identity function, and $<_M$ the lexicographic order on $[\mathbb{N} \times \mathbb{N}]$, i.e., $(a, b) <_{lex} (c, d) \equiv a < c \vee (a = c \wedge b < d)$ where $<$ is the less-than relation on natural numbers. To prove that $T_T^{[[\mathbb{N}\times\mathbb{N}],<_{lex},id]}(\mathtt{ack})$ holds, it suffices to check that for every input pair v_i, leading to any of the recursive-call subexpressions $\mathtt{rec}(e)$ in $\mathtt{ack}$, v_i is such that for every pair v_o satisfying $\varepsilon(\mathtt{ack})(e, v_i, v_o)$, $v_o <_{lex} v_i$.

There are only three recursive calls in the function `ack`, namely: `rec(op1(3,vr))`, `rec(op1(4,vr))`, and `rec(op2(0,vr,rec(op1(4,vr))))`. Each of them determines a case in the proof. For the first subexpression, note that any input value v_i leading to it must be such that $\pi_1(v_i) \neq 0$ and $\pi_2(v_i) = 0$, in order to falsify the guard in the outermost if-then-else and validate the guard in the innermost conditional. Because of the function $O_1(3)$ used to interpret $\texttt{op1}(3, \cdot)$, for every v_o such that $\varepsilon(\texttt{ack})(e, v_i, v_o)$ holds, $\pi_1(v_o)$ must be equal to $\pi_1(v_i) - 1$; hence, $v_o <_{lex} v_i$ holds. For the other recursive-call subexpressions in `ack`, the values v_i that lead to them satisfy $\pi_1(v_i) \neq 0$ and $\pi_2(v_i) \neq 0$. In particular, for the case of `rec(op1(4,vr))`, the function $O_1(4)$ forces any v_o for which $\varepsilon(\texttt{ack})(e, v_i, v_o)$ holds, to be equal to $(\pi_1(v_i), \pi_2(v_i) - 1)$, satisfying $v_o <_{lex} v_i$ as well. Finally, for the values v_i reaching `rec(op2(0,vr,rec(op1(4,vr))))` and because of $O_2(0)$, the first coordinate of v_o must be $\pi_1(v_i) - 1$, which is enough to conclude that $v_o <_{lex} v_i$ holds. Then, $T_T^{[[\mathbb{N}\times\mathbb{N}], <_{lex}, id]}(\texttt{ack})$ holds and, by Theorem 6, $T_\varepsilon(\texttt{ack})$ holds as well. □

The inequalities of the form $v_o <_{lex} v_i$ that are proved in Lemma 5 correspond to the actual termination correctness conditions generated by the PVS type checker for the function `ackermann` defined in Example 1.

4 The Size Change Principle and Calling Context Graphs

4.1 The Size Change Principle

The Size Change Principle (SCP) is another criterion of what it means for a recursive function to terminate. It states that "a program terminates on all inputs if every infinite call sequence (following program control flow) would cause an infinite descent in some data values" [13]. Of course, the definition of "descent" and "data values" here must be qualified in some way, essentially to be some well-founded relation, to make this true. Regardless, the SCP is a particularly useful and concise way to describe termination. The implementation of the SCP inside PVS0 requires a few definitions prior to use.

Definition 15 (*Valid Trace*) Given $\texttt{p} \in \texttt{PVS0}$, and an index set $I \in \{\mathbb{N} \cup \bigcup_i^\infty \mathbb{N}_{\leq i}\}$ a sequence $\mathbf{cc} = \langle \texttt{rec}(e_i), p_i, \mathbf{c}_i\rangle_{i\in I}$ of calling contexts of p, and a sequence of values $\mathbf{v}$ from $\mathscr{V}$ also indexed by I, $\mathbf{cc}$ and $\mathbf{v}$ are said to form a *valid trace of calls* if the following predicate τ holds.[4]

$$\tau_{\texttt{p}}(\mathbf{cc}, \mathbf{v}) \equiv \forall (i-1 \in I) : (\varepsilon_{\pm}(\texttt{p})(\mathbf{c}_{i-1}, \mathbf{v}_{i-1}) \wedge \varepsilon(\texttt{p})(e_{i-1}, \mathbf{v}_{i-1}, \mathbf{v}_i)).$$

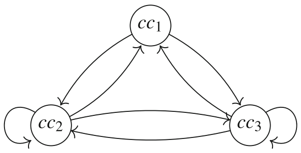
Note that a valid trace can be infinite or finite.

Definition 16 (*Trace-Termination*) A PVS0 program p is said to be *SCP-terminating*, denoted by $T_{SCP}(\texttt{p})$, if there is no infinite valid trace.

Theorem 7 *For all* $\texttt{p} \in \texttt{PVS0}$, $T_\varepsilon(\texttt{p})$ *if and only if* $T_{SCP}(\texttt{p})$.

Proof By Theorem 6 it is enough to prove that $T_T(\texttt{p})$ and $T_{SCP}(\texttt{p})$ are equivalent. Proving $T_{SCP}(\texttt{p})$ given $T_T(\texttt{p})$ is straightforward. To prove the other direction, it is necessary to use $\Omega_{\texttt{p},m}$. Since one has $T_{SCP}(\texttt{p})$, it is possible to provide a relation between parameters

[4] Since $\varepsilon_\pm$ can be straightforwardly extended to lists of polarized expressions, the same symbol is used for both versions along the text. Also, the actual formalization only defines infinite valid traces,



- $cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
- $cc_2 = (ack(m-1, ack(m,n-1)), m \neq 0 \wedge n \neq 0)$
- $cc_3 = (ack(m,n-1), m \neq 0 \wedge n \neq 0)$

Fig. 3 A possible CCG for the Ackermann function

and arguments of recursive calls and prove that it is well-founded. Similarly to the proof of Theorem 6, the closure of this relation is then used to parameterize the function $\Omega_{\mathtt{p},m}$, which provides the height of the tree of evaluation of recursive calls as the needed measure. □

Definition 17 (*SCP-Termination*) Let $<$ be a well-founded relation over $\mathscr{V}$, $\mathtt{SCP}_<(\mathtt{p})$ holds if for all infinite sequence $\mathbf{cc}$ of calling contexts of $\mathtt{p}$ and for every infinite sequence $\mathbf{v}$ of values in $\mathscr{V}$ such that $\tau(\mathbf{cc}, \mathbf{v})$ holds, $\mathbf{v}$ is a decreasing sequence on $<$, i.e., for all $i \in \mathbb{N}$, $v_{i+1} < v_i$.

Theorem 8 *For all $\mathtt{p} \in \mathtt{PVS0}_{\mathscr{V}}$, $T_{SCP}(\mathtt{p})$ if and only if $\mathtt{SCP}_<(\mathtt{p})$ for a well-founded relation $<$ over $\mathscr{V}$.*

4.2 Calling Context Graphs

Calling Context Graphs is a technique that implements SCP [14]. A call sequence in a recursive function (of any reasonable length) will necessarily return to the same "place" in the recursive function more than once. For example, in the Ackemann function, the recursive call guarded by $n = 0$ will be executed many times in a non-trivial example. This Calling Context Graph (CCG) method creates a graph that in essence puts these similar recursive calls into equivalence classes as the vertices, with an edge connecting two recursive calls if one can reach the other. A more precise description of the notion for a PVS0 program is given below.

Definition 18 A *Calling Context Graph* of a PVS0 program $\mathtt{p}$ ($\mathtt{p} \in \mathtt{PVS0}_{\mathscr{V}}$) is a directed graph $G_{\mathtt{p}} = (V, E)$ with a node in V for each calling context in $\mathtt{p}$ such that given two calling contexts of $\mathtt{p}$ $(\mathtt{rec}(e_a), P_a, C_a)$ and $(\mathtt{rec}(e_b), P_b, C_b)$, if

$$\exists (v_a, v_b : \mathscr{V}) : \varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b) \wedge \varepsilon_{\pm}(\mathtt{p})(C_b, v_b),$$

then the edge $\langle (\mathtt{rec}(e_a), P_a, C_a), (\mathtt{rec}(e_b), P_b, C_b) \rangle \in E$.

Note in particular the one way implication in the definition. It does *not* include the converse statement, that an edge must be removed from the graph if there is no valid value pair that relates them. Any graph that obeys this implication is called a *sound* CCG, or just a CCG, for the function. A CCG where every edge that exists is witnessed by some concrete input to the function is called *exact*. While sound graphs are a necessity, an exact graph is not. This is important from a practical implementation standpoint, since the guard for a recursive call can be any boolean function, even an undecidable one. Indeed, the condition on the edges admits a fully connected graph of calling contexts to be considered a sound CCG for the function. For the sake of example, another possible CCG for the Ackermann function as defined in the Example 1 is depicted in Fig. 3, where the calling contexts from Example 3 are abbreviated to improve readability.

The lack of the loop at cc_1 is justified, because there exist no tuples $(a, b), (c, d) \in [\mathbb{N} \times \mathbb{N}]$ such that $\varepsilon_{\pm}(\texttt{ack})(C_{cc_1}, (a, b)) \wedge \varepsilon(\texttt{ack})(e_{cc_1}, (a, b), (c, d)) \wedge \varepsilon_{\pm}(\texttt{ack})(C_{cc_2}, (c, d))$, since this formula can be expanded to $(a \neq 0 \wedge b = 0) \wedge (c = a - 1 \wedge d = 1) \wedge (c \neq 0 \wedge d = 0)$. Note also that the edge from cc_2 to cc_1 could be removed from the graph in the Fig. 3. This is because there are no pairs of naturals $(a, b), (c, d)$ such that $(a \neq 0 \wedge b \neq 0) \wedge (c = a - 1 \wedge d = ack(a, b - 1)) \wedge (c \neq 0 \wedge d = 0)$ since the result of the Ackermann function is always greater than zero. While this is true, it is not a simple property to determine from the function definition without analysis. Moreover, this *sound* but not *exact* calling context graph suffices to prove termination for the function. In practice, deciding which edges are able to be pruned without adding a significant proof burden to a user can be difficult, as the guard for a recursive call can be any boolean function.

The following standard notions from Graph Theory will be used in the definitions below. A *walk* of $G_{\texttt{p}}$ is a sequence $cc_{i_1}, \ldots, cc_{i_n}$ of calling contexts such that for all $1 \leq j < n$ there is an edge between cc_{i_j} and $cc_{i_{j+1}}$. The collection of all walks of a given graph G is denoted by $\mathbf{Walk}_G$. A *circuit* is a walk $cc_{i_1}, \ldots, cc_{i_n}$, with $n > 1$, where $cc_{i_1} = cc_{i_n}$. A *cycle* is an elementary circuit, i.e., a circuit $cc_{i_1}, \ldots, cc_{i_n}$ where the only repeating nodes are cc_{i_1} and cc_{i_n}. The notation $|\mathbf{w}|$ will be used in the following to denote the number of edges in the walk $\mathbf{w}$, and $|G|$ to denote the size of a graph G. Additionally, if $\mathbf{w} = cc_1, \cdots, cc_n$ the expression $\mathbf{w}[a..b]$ will denote the walk $cc_a, \cdots, cc_b$ when $1 \leq a \leq b \leq n$.

Definition 19 Let $\mathscr{M}$ be a *family* of N measures $\mu_k : \mathscr{V} \to M$, with $1 \leq k \leq N$, and $<$ be a well-founded relation over M. A *measure combination* of a sequence of calling contexts $cc_{i_1}, \ldots, cc_{i_n}$ is a sequence of natural numbers $k_1, \ldots, k_n$, with each k_j in the range $1 \leq k_j \leq N$ representing the measure μ_{k_j}, such that for all $1 \leq j < n$, $v, v' \in \mathscr{V}$, $\varepsilon_{\pm}(\texttt{p})(C_j, v) \wedge \varepsilon(\texttt{p})(e_j, v, v')$ implies $\mu_{k_j}(v) \;\; \rhd_j \;\; \mu_{k_{j+1}}(v')$, where $cc_{i_j} = (\texttt{rec}(e_j), P_j, C_j)$ and $\rhd_j \in \{>, \geq\}$. A measure combination is *descending* if at least one $\rhd_j$ is $>$.

Following the example in Fig. 3 and using the family of measures $\mathscr{M} = \{\mu_1, \mu_2\}$ where each of them is a projection of the arguments $\mu_1(a, b) = a$ and $\mu_2(a, b) = b$, the sequence $[1, 2, 2]$ is a descending measure combination for the sequence of calling contexts $[cc_1, cc_3, cc_1]$, since:

(1) $\forall (m, n) \in [\mathbb{N} \times \mathbb{N}],\; m \neq 0 \wedge n = 0 \Rightarrow \mu_1(m, n) \geq \mu_2(m - 1, 1)$, and
(2) $\forall (m, n) \in [\mathbb{N} \times \mathbb{N}],\; m \neq 0 \wedge n \neq 0 \Rightarrow \mu_2(m, n) > \mu_2(m, n - 1)$.

Definition 20 Let $G_{\texttt{p}}$ be a CCG of a PVS0 program $\texttt{p} \in \texttt{PVS0}_{\mathscr{V}}$ and let $\mathscr{M}$ be a family of measures for a well-founded relation $<$ over a type M. The graph $G_{\texttt{p}}$ is said to be *CCG terminating* (denoted by $T_{CCG}(G_{\texttt{p}})$) if for all circuits $cc_{i_1}, \ldots, cc_{i_n}$ in $\mathbf{Walk}_{G_{\texttt{p}}}$ there is a *descending* measure combination $k_1, \ldots, k_n$, with $k_1 = k_n$.

The following case analysis on the possible circuits in the CCG in Fig. 3 reveals that it is terminating. For any given circuit, if the calling contexts cc_1 or cc_2 appear in it, the measure combination formed only by the μ_1 measure is descending, since the projection of the first argument provokes a strict descent for both calling contexts, while for cc_3 the descending flow stands even in a non-strict way. In the remaining case, those circuits formed only by the calling context cc_3 (possibly appearing more than once), a descending measure combination can be constructed using only the projection of the second argument: μ_2.

Theorem 9 *For all* $\texttt{p} \in \texttt{PVS0}_{\mathscr{V}}$, $T_{SCP}(\texttt{p})$ *if and only if* $T_{CCG}(G_{\texttt{p}})$ *for some CCG* $G_{\texttt{p}}$ *of* $\texttt{p}$ *and some family of measures* $\mathscr{M}$.

Since the number of circuits in a CCG is potentially infinite, CCG termination does not directly provide an effective procedure to check termination. The following example shows

$$\begin{tikzcd} cc_2 \arrow[loop left] \arrow[r, bend left] & cc_1 \arrow[l, bend left] \arrow[r, bend left] & cc_3 \arrow[l, bend left] \arrow[loop right] \end{tikzcd}$$

- $\mu_1(n,m,s) = n$
- $\mu_2(n,m,s) = m$
- $\mu_3(n,m,s) = s$
- $\mu_4(n,m,s) = \max(n,m)$
- $\mu_5(n,m,s) = \max(n,m)+2$

- $[cc_2,cc_2] : [1,1]$
- $[cc_3,cc_3] : [2,2]$
- $[cc_2,cc_1,cc_2] : [5,5,4]$
- $[cc_1,cc_2,cc_1] : [5,4,4]$
- $[cc_3,cc_1,cc_3] : [5,5,4]$
- $[cc_1,cc_3,cc_1] : [5,4,4]$

Fig. 4 A possible CCG, a family of measures, and descending combinations for the definition in Example 5

cc_1	$\mu_1(m-1,1)$	$\mu_2(m-1,1)$
$\mu_1(m,0)$	$>$	$\geq$
$\mu_2(m,0)$	$\leq$	$<$

cc_3	$\mu_1(m,n-1)$	$\mu_2(m,n-1)$
$\mu_1(m,n)$	$\geq$	?
$\mu_2(m,n)$	?	$>$

Fig. 5 Summary of measure applications for the calling contexts cc_1 and cc_3 in the CCG from Fig. 3

that even though the number of cycles in a graph, i.e., its non-repeating circuits, is indeed finite, it is not enough to check for decreasing measure combinations on them.

Example 5 Consider the definition on natural numbers stated below, where $0?(n)$ and $\neg 0?(n)$ are abbreviations for $1 \dot{-} n$ and $1 - 0?(n)$ respectively, and $\dot{-}$ is the saturated subtraction defined as $a \dot{-} b = max(a - b, 0)$.

$$f(n,m,s) = \begin{cases} f(1 + \neg 0?(\mathrm{s}) * (m+n),\, 1 + 0?(\mathrm{s}) * (m+n),\, 0?(\mathrm{s})) & \text{if } n = 0 \text{ or } m = 0 \\ f(n-1, m, 0) & \text{otherwise and } s = 0 \\ f(n, m-1, 1) & \text{otherwise} \end{cases}$$

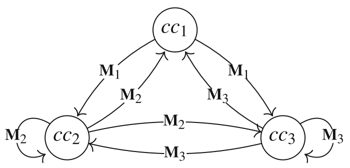
This definition can be expressed as a `PVS0` program. Figure 4 shows a valid CCG for f, where the calling contexts are numbered in the order they appear in the definition. The family of measures presented there can be used to form descending combinations for each cycle in the graph, as shown in the same Figure. Nevertheless, f is not terminating since, for instance, the inputs $1, 1, 0$ cause the following chain of calls: $f(1,1,0) \to f(0,1,0) \to f(1,2,1) \to f(1,1,1) \to f(1,0,1) \to f(2,1,0) \to f(1,1,0) \to \cdots$, over the circuit $cc_2, cc_1, cc_3, cc_3, cc_1, cc_2, cc_2, \ldots$. This explains why Definition 20 cannot be stated in terms of cycles.

5 Matrix-Weighted Graphs

Matrix-Weighted Graphs (MWG) is a technique that checks descending measure combinations in CCG using an algebra over matrices [4]. Given a CCG G_{p} for a *PVS0* program p, a family of measures $\mathscr{M}$, and a calling context $cc \in G$, the main idea behind MWG is to summarize the relation between the application of a measure μ_a on the arguments of p and the application of a measure μ_b on the arguments of the recursive call in cc for every possible pair of measures $\mu_a, \mu_b \in \mathscr{M}$.

Figure 5 shows a way in which such information can be summarized for two of the calling contexts in the CCG depicted in Fig. 3.

There, each row of the tables represents the application of one of the measures μ_1 or μ_2 as stated above on the arguments of the function. Similarly, each column represents the application of one of the measures on the arguments of the recursive call in the corresponding calling context. Every symbol inside the table informs the order relation between the result of the application denoted by the row and the one denoted by the column when it can be stated



$$\mathbf{M}_1 = \begin{bmatrix} 1 & 0 \\ -1 & -1 \end{bmatrix} \; \mathbf{M}_2 = \begin{bmatrix} 1 & -1 \\ -1 & -1 \end{bmatrix} \; \mathbf{M}_3 = \begin{bmatrix} 0 & -1 \\ -1 & 1 \end{bmatrix}$$

Fig. 6 A MWG for the PVS0 program for the Ackermann function

for any possible input value fulfilling the path conditions of the calling context. A question mark is used otherwise.

Capturing these kinds of interactions among the members of a family of N measures can be done using matrices of dimension $N \times N$ and values in $\{-1, 0, 1\}$. Since the only relations that are significant to decide CCG termination are $\geq$ and $>$, the values 0 and 1 are used to represent them, while the value -1 is used to represent an increase on the measure of the arguments or the impossibility of stating any relation between them (signaled with a question mark in Fig. 5). The type of the matrices is denoted by $\mathbb{M}_{\mathbf{3}}^{N}$. Similar to the pruning of edges of the CCG, the construction of *sound* measure matrices, in the sense that any 0 or 1 entries correspond to an inequality that must be proven, is necessary. A measure matrix with all entries set to -1 is sound, but does not capture any information about measures decreasing. Being able to effectively determine some entries that can be marked 0 or 1, and prove the associated inequality, is undecidable in general.

Definition 21 (*Matrix Weighted Graph*) Let $\mathtt{p}$ be a *PVS0* program in $\mathtt{PVS0}_{\mathscr{V}}$ and $\mathscr{M}$ be a family of N measures $\{\mu_i\}_{i=1}^{N}$. A *matrix-weighted graph* $W_{\mathtt{p}}^{\mathscr{M}}$ of $\mathtt{p}$ is a CCG $G_{\mathtt{p}} = (V, E)$ of $\mathtt{p}$ whose edges are *correctly* labeled by matrices in $\mathbb{M}_{\mathbf{3}}^{N}$.

An edge $(cc_a, cc_b) \in E$ is said to be *correctly labeled* by a matrix $\mathbf{M}_{ab}$ when for all $1 \leq i, j \leq N$,

– if $\mathbf{M}_{ab}(i, j) = 1$, then for any pair of values $v_a, v_b \in \mathscr{V}$,

$$\varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b) \text{ implies } \mu_i(v_a) \ > \ \mu_j(v_b);$$

– if $\mathbf{M}_{ab}(i, j) = 0$, then for any pair of values $v_a, v_b \in \mathscr{V}$,

$$\varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b) \text{ implies } \mu_i(v_a) \ \geq \ \mu_j(v_b).$$

An entry $\mathbf{M}_{ab}(i, j) = -1$ provides no information about v_a, v_b with respect to μ_i and μ_j.

Similar to the pruning of edges of the CCG, the construction of *sound* measure matrices, in the sense that any 0 or 1 entries correspond to an inequality that must be proven, is necessary. As mentioned, a measure matrix with all entries set to -1 is sound, but does not provide any information about measures decreasing. Being able to determine some entries that can be marked 0 or 1, and prove the associated inequality, is undecidable in general. A measure matrix where relationships between measures are marked with the highest value that is true for every input (i.e., a 1 value for measure pairs that strictly decrease on all inputs, a 0 value measure pairs that weakly decrease on all inputs with at least one set of inputs witnessing equality, and a -1 value for measure pairs with a set of inputs witnessing increase) is called *exact*. Luckily, termination can often be proven with *sound* but not *exact* measure matrices.

Figure 6 depicts a possible MWG for the PVS0 program implementing the Ackermann function.

The next step is to combine these matrices in order to check if a suitable measure combination as stated in Definition 19 exists for two consecutive calling contexts in the graph. The method for combining the information summarized in the matrices is to multiply them as usual, but redefining the binary operations on the entries. Let $x, y \in \{-1, 0, 1\}$, the addition of x an y is the maximum between them, $x + y = \max(x, y)$, while the multiplication is shown below.

$$x \times y = \begin{cases} -1 & \text{if } \min(x, y) = -1, \\ 1 & \text{if } \min(x, y) \geq 0 \wedge \max(x, y) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 22 (*Weight of a Walk*) Let $\mathtt{p}$ be a PVS0 program, $W_{\mathtt{p}}$ a MWG for $\mathtt{p}$, and $\mathbf{w_i} = cc_{i_1}, \ldots, cc_{i_n}$ a walk in such graph, the *weight* of $\mathbf{w_i}$, noted by $w(\mathbf{w_i})$, is defined as $\Pi_{j=1}^{n-1} \mathbf{M}_{i_j i_{j+1}}$. A weight $w(\mathbf{w_i})$ is *positive* if there exists $1 \leq i \leq N$ such that $w(\mathbf{w_i})(i, i) > 0$.

Example 6 Continuing the example in Fig. 6, the weights for walks $\mathbf{w_{1,3}} = cc_1, cc_3$ and $\mathbf{w_{2,3}} = cc_2, cc_3$ are shown below. Both of them are positive.

$$w(\mathbf{w_{1,3}}) = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \qquad w(\mathbf{w_{2,3}}) = \begin{bmatrix} 1 & -1 \\ -1 & -1 \end{bmatrix}$$

The lemma below states a useful property about walk weights.

Lemma 6 *Let $W_{\mathtt{p}}$ be an MWG for a PVS0 program $\mathtt{p}$ and $\mathbf{w} = cc_1, \cdots, cc_n$ be a walk of $W_{\mathtt{p}}$, then $w(\mathbf{w}) = w(cc_1, \cdots, cc_i) \times w(cc_i, \cdots, cc_n)$.*

As in the case of the calling context graphs, a walk in a MWG represents a trace of recursive calls. Hence, a circuit denotes a trace ending at the same recursive call where it starts. In line with the notion of CCG termination, a MWG is considered *terminating* when, for every possible circuit, the matrix representing its weight has at least one positive value in its diagonal.

Definition 23 (*Matrix-Weighted Graph Termination*) Let $\mathtt{p}$ a PVS0 program and let $W_{\mathtt{p}}$ be a MWG of $\mathtt{p}$. The graph $W_{\mathtt{p}}$ is said to be *MWG terminating* (denoted by $T_{MWG}(W_{\mathtt{p}})$) when for every circuit $\mathbf{w_i}$ of $W_{\mathtt{p}}$, $w(\mathbf{w_i})$ is positive.

The equivalence between the notions of termination for CCG and MWG is stated by Theorem 10 below.

Theorem 10 *Let $\mathscr{M}$ be a family of N measures for a well-founded relation $<$ over a type M. For all $\mathtt{p} \in \mathtt{PVS0}_{\mathscr{V}}$, $T_{CCG}(C_{\mathtt{p}}{}^{\mathscr{M}})$ for some CCG $C_{\mathtt{p}}{}^{\mathscr{M}}$ if and only if $T_{MWG}(W_{\mathtt{p}}{}^{\mathscr{M}})$ for some MWG $W_{\mathtt{p}}{}^{\mathscr{M}}$.*

Proof This theorem follows from the fact that circuits in $W_{\mathtt{p}}$, built from $G_{\mathtt{p}}$ using the same measures, have positive weights if and only if there exist corresponding descending measure combinations. This property is proved by induction in the length of circuits in $G_{\mathtt{p}}$. □

5.1 Bounding Circuit Length

As pointed out in the previous section, a digraph such as any CCG or MWG can have infinitely many circuits. Nevertheless, since the information used to check MWG termination is the weight of the circuits and, for a fixed number N of measures, there are only finitely many possible weights, a bound on the length of the circuits to be considered can be safely imposed as shown in the lemma below.

Lemma 7 *Let* $\mathtt{p}$ *be a PVS0 program and* $W_{\mathtt{p}}$ *a MWG for it. If for all circuit* $\mathbf{w}$ *in* $W_{\mathtt{p}}$ *such that* $|\mathbf{w}| \le |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, $w(\mathbf{w})$ *is positive, then* $W_{\mathtt{p}}$ *is MWG terminating.*

Proof In order to prove $T_{MWG}(W_{\mathtt{p}})$, it is necessary to show that every circuit of $W_{\mathtt{p}}$ has positive weight. For every circuit $\mathbf{w} = cc_1, \cdots, cc_n$ of $W_{\mathtt{p}}$, if $n \le |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, then $w(\mathbf{w})$ is positive by hypothesis. Otherwise, it can be proved that there exists another circuit $\mathbf{w}'$ such that $w(\mathbf{w}) = w(\mathbf{w}')$ and $|\mathbf{w}'| < |\mathbf{w}|$. By hypothesis, $w(\mathbf{w})'$ is positive and so $w(\mathbf{w})$ is positive also. Hence by an inductive argument, the result will hold.

The existence of the circuit $\mathbf{w}'$ is established using a pigeonhole principle argument. Given the walk $\mathbf{w}$ of length $n > |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$ in $W_{\mathtt{p}}$, construct the sequence of ordered pairs $\langle (cc_i, w(cc_1, \cdots, cc_i)) \rangle_{i=1}^{n}$, where for each $1 \le i \le n$, the vertex cc_i is the i^{th} vertex in $\mathbf{w}$ and it is paired with the weight of the prefix of $\mathbf{w}$ of length i (essentially the pair is the vertex, and the accumulated weight of the walk to that point). By the pigeonhole principle, it can be seen that there cannot exist more than $|W_{\mathtt{p}}| \cdot 3^{N^2}$ of these pairs that are all distinct. Since $n > |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, there are two indices i, j such that $(cc_i, w(cc_1, \cdots, cc_i)) = (cc_j, w(cc_1, \cdots, cc_j))$ and $i \ne j$. Without loss of generality, it can be assumed that $i < j$. Then, consider the walk $\mathbf{w}' = cc_1, \cdots, cc_{i-1}, cc_j, cc_{j+1}, \cdots, cc_n$. Note that it is a circuit, since $cc_i = cc_j$ and $cc_1 = cc_n$. Also note that the length of $\mathbf{w}'$ is shorter than $\mathbf{w}$, since the path from cc_i to cc_{j-1} is removed, which contains at least the vertex cc_i. To calculate the weight of $\mathbf{w}'$, first it should be noted that, by Lemma 6, $w(cc_1, \cdots, cc_j, cc_{j+1}, \cdots, cc_n) = w(cc_1, \cdots, cc_{i-1}, cc_j) \times w(cc_j, cc_{j+1}, \cdots, cc_n)$. Since the vertex cc_i is the same than the vertex cc_j and by our hypothesis, $w(cc_1, \cdots, cc_i)$ is equal to $w(cc_1, \cdots, cc_j)$, then $w(\mathbf{w}') = w(cc_1, \cdots, cc_j) \times w(cc_j, cc_{j+1}, \cdots, cc_n)$, which by Lemma 6 again is equal to $w(\mathbf{w})$. Conceptually, the process finds a circuit at vertex cc_i that has the same weight when entering the circuit as when leaving, and hence can be "cut out" without changing the weight of the original walk.

If the length of $\mathbf{w}'$ is at most $|W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, the result is proven. Otherwise, the procedure can be repeated until the circuit reaches the claimed length.

□

Note that this bound can be turned into a brute-force algorithm for determining the entire collection of weights taken on by the circuits in $W_{\mathtt{p}}$. First, enumerate the circuits of length at most $|W_{\mathtt{p}}| \cdot 3^{N^2} + 1$, building them using the elementary cycles of $W_{\mathtt{p}}$. This can be computed by Tarjan's algorithm [19], or other similar know methods. Once these are in hand, calculate the weight of the circuit for each one. Unfortunately, for even very small graphs and measure matrix sizes, the circuit length bound in Lemma 7 can be large. The number of circuits of this length is even worse, generally being exponential in the number of elementary cycles.

Example 7 Continuing Example 6. As an example, consider the matrix weighted graph for the Ackermann function in Fig. 6. With three vertices, and two measures, the circuit bound is hence $3 \cdot 3^{2^2} + 1 = 244$. Consider only the portion of the graph containing vertices cc_1 and cc_2, and further, only circuits starting and ending at cc_2. There are two elementary cycles, which are the loop at cc_2 and the two-edge path from cc_2 to cc_1 and back to cc_2. The number of different circuits of length k using *just these two cycles starting from* cc_2 is then F_k, the kth Fibonacci number.[5] Hence the number of circuits at cc_2, using only these two cycles, of

[5] This simple exercise is left to the reader.

length at most 244 is

$$\sum_{k=1}^{244} F_k = 115205841188445478830259303420656877245267403732 5127.$$

Computing these circuits and their weights would take an untenable amount of time and resources. So while Lemma 7 implies a basic algorithm, it is clearly not the most efficient one. It should also be noted that the method described does not use any of the structure of the measure matrices, simply relying on the number of them being finite.

5.2 Dutle's Procedure

Lemma 7 states that there is a bound on the length of circuits that must be examined to guarantee that all circuits have positive weight in a matrix-weighted graph. The proof of this bound provides both a guidepost as to how an algorithm should run (by examining successively longer circuits), and a guarantee of termination for an algorithm that does so. The particular algorithm for doing so, described below, is referred to as Dutle's procedure.

Given a MWG ${W_{\mathtt{p}}}^{\mathscr{M}} = (V, E)$ on a family of N measures $\mathscr{M}$ for a PVS0 program $\mathtt{p}$, the general idea of this procedure is to build sequentially a family of functions $f_i : V \to \mathbf{list}[\mathbb{M}_{\mathbf{3}}^N]$ with $0 \leq i \leq |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$. These functions are such that for each vertex $cc \in V$, and every circuit $\mathbf{w}$ in ${W_{\mathtt{p}}}^{\mathscr{M}}$ starting at cc where $|\mathbf{w}| <= i$, there is a weight $\mathbf{M} \in f_i(cc)$ for which $\mathbf{M} \leq w(\mathbf{w})$. Here, the inequality on measure matrices is taken pointwise, noting that the collection of $N \times N$ measure matrices form a partially ordered set under this inequality. For correctness, it is also required that any $\mathbf{M} \in f_i(cc)$ is actually the weight of some circuit at cc.

If for some i there is a vertex cc and a weight $\mathbf{M}$ such that $\mathbf{M} \in f_i(cc)$ and $\mathbf{M}$ is not positive, then it can be concluded that ${W_{\mathtt{p}}}^{\mathscr{M}}$ is not terminating, since there is a circuit whose weight is not positive. On the other hand, the algorithm can be stopped if it reaches the point where $i = |W_{\mathtt{p}}| \cdot 3^{N^2} + 1$ with positive matrices in the range of $f_i(cc)$ for each i, and ${W_{\mathtt{p}}}^{\mathscr{M}}$ can be safely declared as terminating thanks to Lemma 7. To avoid having to continue the computation to this bound, another stopping criterion is introduced, which is in fact always reached prior to attaining the circuit bound.

Figure 7 depicts a pseudo-code for Dutle's procedure. The majority of the computation takes place in the function **expandPartialWeight**. It takes as parameter f_i, which is the current collection of lists, specific to each vertex, which contain the measure matrices encountered as a weight of all circuits of length at most i. When provided with a walk $\mathbf{w}$ as input, **expandPartialWeight**(f_i) calculates a new list of measure matrices, which contains the measure matrices of every walk that can be obtained by walking along $\mathbf{w}$, and at every vertex, taking some circuit of length at most i. The function achieves this by combining weights from the vertex lists on the walk with weights from the edges of the walk. At the first vertex v_0 of $\mathbf{w}$, the list from f_i is taken. Each element of this list is multiplied by the weight of the edge leading to the second vertex v_1. The resulting list is then combined with the list from v_1 using the function **pairwiseMultiplication** which, given two lists l_1, l_2 of matrices in $\mathbb{M}_{\mathbf{3}}^N$ returns the list resulting from the pairwise multiplication of the elements in those lists. This procedure is then repeated until the end of the walk. Note that since the values from f_i for each vertex contain weights from cycles of length at most i (including $i = 0$), this procedure finds the weight of any walk $\mathbf{w}'$ that traverses $\mathbf{w}$, but at each vertex, chooses and traverses any cycle of length at most i.

```
**terminating**?($W_p$: MWG): bool =
  LET $f_0 \leftarrow$ **expandWeightLists**($W_p, \lambda(v: V_{W_p})$ : **null**)
  IN **terminatingAt**?($W_p$, 0, $f_0$)

**terminatingAt**?($W_p$: MWG, $i$: $\mathbb{N}$, $f_i$ : [$V_{W_p} \to$ **list**[$\mathbb{M}_3^N$]]): bool =
 $i \geq |W_p| \cdot 3^{N^2} + 1$ OR
 LET $f_{i+1} \leftarrow$ **expandWeightLists**($W_p$, $f_i$) IN
 IF $\exists$ $(cc \in V_{W_p}, \mathbf{M} \in f_{i+1}(cc)) : \neg$ **positive**?($\mathbf{M}$) THEN FALSE
 ELSE $f_i$ = $f_{i+1}$ OR **terminatingAt**?($W_p$, $i+1$, $f_{i+1}$) ENDIF

**expandWeightLists**($W_p$: MWG, $f_i$ : [$V_{W_p} \to$ **list**[$\mathbb{M}_3^N$]]): [$V_{W_p} \to$ **list**[$\mathbb{M}_3^N$]] =
 $\lambda(v: V_{W_p})$: **map**(**expandPartialWeight**($f_i$), **allCyclesAt**($W_p$,$v$))

**expandPartialWeight**($f_i$ : [$V_{W_p} \to$ **list**[$\mathbb{M}_3^N$]]): [**Walk**$_{W_p} \to$ **list**[$\mathbb{M}_3^N$]] =
 $\lambda$($\mathbf{w}$ : **Walk**$_{W_p}$):
   LET $l$ $\leftarrow$ **cons**(**id**$_\times$, $f_i$($\mathbf{w}$[0]))
   IN IF $|\mathbf{w}|$ = 1 THEN $l$
      ELSE LET $l_1 \leftarrow$ **map**($\lambda$ ($\mathbf{M}$: $\mathbb{M}_3^N$): $\mathbf{M}$ $*$ $w(\mathbf{w}[0..1])$)($l$),
               $l_2 \leftarrow$ **expandPartialWeight**($\mathbf{w}$[1 .. $|\mathbf{w}|-1$], $f_i$)
           IN **pairwiseMultiplication**($l_1$,$l_2$) ENDIF
```

Fig. 7 Dutle's procedure to check termination on matrix-weighted graphs

The wrapper function **expandWeightLists** applies the **expandPartialWeight** to the collection of all elementary cycles at each vertex, which is computed using the auxiliary function **allCyclesAt**(G, v). Note that any circuit of length at most $i + 1$ is either an elementary cycle, or can be decomposed as an elementary cycle with circuits of length at most i attached to each vertex of the elementary cycle. Hence the function **expandWeightLists** computes f_{i+1} given its predecessor f_i. It is worth noting that the claimed invariant of f_{i+1} is that is contains weights of all circuits of *length* at most $i + 1$, but in fact it computes something more. It calculates the weights of all circuits with some loosely defined notion of *circuit complexity* at most $i + 1$. Because this notion was not crucial to the procedure, it is not defined precisely or analyzed formally in this development.

The function **terminatingAt**? recursively computes the desired lists f_i, and also implements the stopping conditions. The first stopping condition is $i \geq |W_p| \cdot 3^{N^2} + 1$, which allows the use of Lemma 7 to guarantee that the procedure itself will terminate. The second condition determines if there is a cycle of negative weight using the negation of the function **positive**?($\mathbf{M}$), which checks if a matrix $\mathbf{M}$ is positive in the sense of Definition 22. In this case, the procedure exits with false, indicating that a non-positive cycle was detected. The third condition implements the recursive step, as well as the innocuous-looking check $f_i = f_{i+1}$. This check on the *saturation* of f_i serves two (related) purposes. The main reason for the check is that once a single computation step does fails to change f_i, any further iterations will also leave it unchanged, and so it follows that $f_i = f_{|W_p| \cdot 3^{N^2}+1}$. This can happen based on many things, including the structure of the particular calling context graph, the measures chosen, and the calculations of the measure comparisons. Additionally, this saturation accounts for the inherent looseness in the bounds used to guarantee termination of the procedure. The bound for Lemma 7 is based on a very rudimentary pigeonhole principle argument, and a closer examination could likely lower this value. Similarly, as mentioned above, the calculation of f_i actually bounds circuits of a certain *complexity* that also ensures length, and further analysis of this complexity would likely show it as *strictly* encompassing length.

Finally, the outer call to this procedure is **terminating**?, which initializes the set of lists to be empty for f_0, since a walk of length 0 has no weight.

Other elements in the pseudocode include **cons**(x, l), which denotes the list constructed from the element x and the list l, **null** which denotes the empty list, and **map**(f, l), used to denote the list formed by the application of the function f to each element in l.

Dutle's Procedure is a sound and complete procedure to decide the positive weight of all circuits in a matrix-weighted graph and hence to check termination on an MWG. This procedure has been formally verified in PVS as part of this work.

Several aspects of this procedure could be further optimized. For instance, both execution time and storage space can be improved. The function **expandWeightLists** enlarges the lists in the range of each f_{i+1} (with respect to its predecessor f_i), while it is enough to keep such lists minimal, for instance by adding a new weight $\mathbf{M}$ to a list l only if there are no $\mathbf{M}'$ in l already such that $\mathbf{M}' \leq \mathbf{M}$. This is true because the measure matrices, along with the multiplication operation defined on them, form a semi-group where the multiplication operation respects the partial order on the matrices induced by the pointwise inequality relation. This implies that keeping a minimal set of weights achieved at a point in the process for each vertex is sufficient.

As implied by the discussion above, the check for $i \geq |W_{\mathrm{p}}| \cdot 3^{N^2} + 1$ could also be removed, since the saturation of f_i is guaranteed by this point. Indeed, in a more general setting, the measure matrices can be replaced by any partially ordered semi-group, and the notion of a "non-positive" weight can be replaced with an arbitrary filter. Provided that the partial order has finite width, and the descending chain condition, the same procedure is guaranteed to terminate.[6] A formal proof of this more general procedure has not been undertaken, as the specific case of measure matrices was enough for the work under consideration.

As a test, the implementation of **terminating**? inside this development was temporarily equipped to count the number of interactions of **terminatingAt**? required to finish computation, either by saturation or by reaching the cycle length bound. Using the matrix weighted digraph from Fig. 6 developed for the Ackermann function, only 2 iterations were required for saturation of the lists, while the cycle length bound is 244.

5.3 Automating Termination Analysis

The notion of Matrix Weighted Termination can be used to define a full procedure to automatically prove termination of certain recursive functions in PVS. Such a procedure consists of the steps described below.

1. Extract the calling contexts from the PVS program definition. The set of calling contexts is finite and can be extracted from the program by syntactic analysis.
2. Generate a sound CCG for the program.
   - A fully connected CCG is *sound* (the more edges the more inefficient the method, and in fact superfluous edges may prevent a proof of termination entirely).

[6] Essentially, the proof that the procedure terminates is that at each step, the list f_i at a vertex will contain (at most) a single minimal element from a chain cover of the poset. The number of such elements is bounded due to the finite width, and either these elements stabilize, or decrease according to the poset order. The DCC shows that this cannot happen infinitely often.

  - The theorem prover itself can be used to *soundly* remove edges from the graph, i.e., an edge cc_a, cc_b can be removed if the condition $\forall(v_a, v_b : \mathscr{V}) : \varepsilon_{\pm}(\mathrm{p})(C_a, v_a) \wedge \varepsilon(\mathrm{p})(e_a, v_a, v_b) \Rightarrow \neg\, \varepsilon_{\pm}(\mathrm{p})(C_b, v_b)$ can be discharged.
  - In order to select measures to form the family $\mathscr{M}$, the following heuristics can be used.
    - The order relation $<$ over natural numbers is usually a good starting point.
    - Since CCG allows for a family of measures, it is *sound* to add as many measures as possible (of course the more measures the more inefficient the method).
    - Predefined functions can be used, e.g., parameter projections (in the case of natural numbers), natural size of parameters (in the case of data types), maximum/minimum of parameters, etc. More complex recursions may need heuristics based on static analysis.

3. Construct a MWG for the program based on the CCG defined in the previous step in the following way: all edges starting in a given calling context cc_a can be labeled with the same matrix $\mathbf{M}_a$. It is *sound* to set all its entries to -1. The theorem prover can then be used to *soundly* set the entries in $\mathbf{M}_a(i, j)$ to either 0 or 1 as follows,
   - If $\vdash \forall(v_a, v_b : \mathscr{V}) : \varepsilon_{\pm}(\mathrm{p})(C_a, v_a) \wedge \varepsilon(\mathrm{p})(e_a, v_a, v_b) \Rightarrow \mu_i(v_a) > \mu_j(v_b)$ can be proved, set $M_a(i, j)$ to 1.
   - If $\vdash \forall(v_a, v_b : \mathscr{V}) : \varepsilon_{\pm}(\mathrm{p})(C_a, v_a) \wedge \varepsilon(\mathrm{p})(e_a, v_a, v_b) \Rightarrow \mu_i(v_a) \geq \mu_j(v_b)$ can be proved, set $M_a(i, j)$ to 0.
4. Use Dutle's procedure to check termination on the MWG.

Besides the pure theoretical aspect of this work, one of its long-term goals is to increase the automation of termination analysis of user-defined functions in PVS. A concrete implementation and integration of this procedure for users of PVS has not yet been achieved. Such an implementation could take several possible variations, which can be grouped into two main lines.

Integration into the PVS core: Integration of the MWG termination criteria into the PVS prover itself poses several technical and philosophical issues, depending on how deeply the procedure is embedded into PVS. As noted in Sect. 3, the PVS typechecker currently generates termination TCCs that are essentially Turing termination criteria on a user-provided well-founded order. In the (arguably) most invasive embedding of the procedure, the typechecker could be altered to offer an MWG termination TCC, where the PVS core essentially does all of the steps of the procedure outlined above, including pruning of edges, choosing measures, and executing the procedure. This route offers the most tighly integrated version of MWG termination, but would add a significant amount of code to the PVS base. This option has the highest probability of introducing unsoundness in PVS through some bug in the implementation of the procedure, with modest improvement to termination automation.

A somewhat less invasive integration into the PVS prover would involve having the typechecker offer parametric MWG criteria that the user would instantiate. The typechecker would still be required to extract the calling context from a program, but instead of performing steps 2,3, and 4 from above, it would generate a TCC positing the *existence* of such structures. The user would be required to input and prove the soundness of the CCG, choose measures to be used in the MWG, and prove the validity of the entries of the measure matrices. While this is certainly burdensome on the user, several existing strategies are already available in the MWG development described here that automate many of these steps in a sound way. One possible downside to such an implementation woud be the need to include many of the PVS theories that are used in this development in the prelude libraries. On the positive side, due to the smaller impact on the PVS core, this route is less likely to introduce unsoundness.

A minimally invasive integration in the PVS prover may provide functionality for MWG termination also. Currently PVS takes as *part of the specification* a keyword or a function that is used to generate the termination TCC. The user provides (or points to) the well-founded measure in the specification, and the TCCs ask to prove that it decreases on every recursive call. Keeping the type checker essentially as it, the use of MWG termination can be facilitated by moving the specification of this well-founded measure *into the TCC.* Assuming that steps 1-4 above are able to be performed in the prover through strategies or minimal user interaction (discussed below), the developments described in this paper, particularly the proof that MWG termination implies Turing termination, can be used to prove the existence of such a well-founded measure. This method of incorporating MWG termination into the PVS core is perhaps the most promising, since it requires the least amount of change to the system, and even gives flexibility to incorporate future methods of termination analysis.

The developers of PVS are aware of the termination work detailed here, and are receptive to the idea of possible incorporation of MWG termination in future releases. In the mean time, another route is being explored which is non-invasive to PVS.

Using MWG termination external to the PVS core: An alternative method for using MWG termination in PVS involves not altering the theorem prover at all, instead externally processing a recursive specification in a way that can apply the methods described. This method is currently being investigated as a precursor to the full integration in PVS described above.

Currently, the MWG termination work described here can be used in PVS to prove some termination TCCs that are generated. This process involves several steps. First, a version of the recursive function is manually specified in the language of PVS0. This includes defining a type that can contain the inputs to the function, and embody *true* and *false* values inside PVS0, and also defining the operations and structure of the function. The measures used on the function inputs are specified, and the calling context graph is created and proven sound. As noted above, a complete graph is always sound, but pruning may make the graph more effective. Measure matrices are defined, and assigned to the CCG, where they are required to be proven sound for their assignment. Lemmas are written to embody that this structure proves termination, and that the recursive function has a well founded measure. These can be proven using the Dutle's procedure, and the proof that MWG termination implies measure termination.

This all must occur in the PVS file logically prior to the actual recursive function definition, so that the measure which is proven to exist can be used in the actual recursive defintion. Even with this complete, when using the measure, it must be shown that the measure which applies to the PVS0 function applies to the normal specification, by proving essentially that the two functions are equivalent.

Much of the *proving* in this process can, and has, been automated. For example, there is a PVS strategy that can prove the functional equivalence of a given function with a PVS0 counterpart, a strategy that will take a given MWG and run Dutle's procedure to determine if the structure proves termination, and a strategy that will prove the existence of a well-founded measure if the MWG proves termination.

The more difficult part of automating the procedure is in the specification. Particularly the generation of the PVS0 function, the specification of a sound, pruned CCG, and the specification and assignment of sound measure matrices. Some progress has been made on automating the PVS0 function specification for PVS functions with some particular branching structure, and limited types for input variables, but the general case is still elusive. Currently only the fully connected CCG is able to be automatically generated and proven correct. Some concepts and heuristics for determining which edges of a CCG must be kept and which edges can be proven valid to remove are being investigated. Similarly, choosing measures, and generating and proving that measure matrices are sound is still under development.

5.4 Applicability

While an automatic termination procedure is still in development, As a way to evaluate how effective an integration of a termination analysis procedure based on MWG into the PVS workflow might become, a survey of the recursive definitions declared in PVS is presented below.

NASALib is currently the largest publicly-available collection of PVS formalizations. It was developed collectively by a large number of researchers and practitioners over three decades. It contains more than fifty different libraries covering a wide spectrum of topics ranging from fundamental results on, for example, number theory, linear algebra, probability, and graph theory among others, to advanced and specific numerical methods based on Bernstein polynomials, Tarski and Sturm's Theorems. It is currently curated and maintained by the NASA Langley Formal Methods Team and is expanded year after year based on internal and external contributions.

Table 1 presents detailed information about the recursive definitions declared in NASALib. The first column ("recursive definitions") shows the number of recursive declarations in each library. The declarations are further classified in four groups according to the measure function provided for each one of them.

The first group contains those declarations in which corresponding measures are expressed just as one of the arguments. These are the simplest cases of recursion in which, for example, a natural number is decremented or some element is removed from an inductively-defined data structure in each recursive call. The function to calculate the length of a list or Fibonacci's *Backward Trial Division* procedure to check the primality of a given natural number are canonical citizens of this group. The well-founded relation used to state the termination-related proof obligations in these cases is the natural order on the type of the argument. For example, the less-than relation on natural numbers or the structural inclusion in the case of inductive data types.

The second group, depicted in the third column of the table, is composed of the recursive definitions in which associated measures are expressed as the application of some function denoting the size of just one of the arguments, in particular, the cardinality of a finite set; these sets are not defined inductively in PVS but rather by means of their characteristic predicate.

The measures in the third group are denoted by more complex expressions, involving at least two of the arguments of the recursive function. This group is composed of two kinds of measures: subtraction between two arguments of numeric type, and the sum of the sizes of two arguments with inductively defined data structure types. In the former case, one of the arguments is incremented in each call while the other one is used as a bound for the growth. In the latter, each recursive call reduces the size of at least one of the two arguments involved in the measure.

The fourth group is populated by declarations associated with ad-hoc measures defined by syntactically complex expressions involving cases analysis, the application of several functions, or relating more than two arguments. These are the most complicated of the termination measure classes, which have little hope for automation. Figure 8 shows an example of a function from this group: a recursive definition of the Newton approximation for the square

Table 1 Quantitative information on recursive definitions in NASALib

Library Total	Recursive Definitions **480**	1-argument Projection **351**	Size Function **11**	2-args Comb **50**	Ad-hoc Complex **68**
structures	48	34	2	7	5
sorting	42	33	0	6	3
PVS0	32	25	0	1	6
affine_arith	31	24	0	7	0
reals	30	23	0	4	3
Bernstein	27	20	0	6	1
Tarski	27	21	0	1	5
matrices	24	13	1	5	5
digraphs	22	16	2	0	4
TRS	21	17	0	0	4
Sturm	17	10	0	2	5
co_structures	16	13	0	0	3
interval_arith	14	12	0	1	1
while	14	14	0	0	0
CCG	13	11	0	0	2
fast_approx	11	4	0	0	7
exact_real_arith	10	6	0	1	3
analysis	7	6	1	0	0
ACCoRD	7	5	0	0	2
ints	6	2	1	1	2
orders	6	5	1	0	0
aviation	6	3	0	0	3
Riemann	6	5	0	1	0
numbers	5	3	0	0	2
algebra	5	3	0	2	0
sets_aux	4	3	1	0	0
trig	4	4	0	0	0
vectors	4	0	0	4	0
float	4	2	0	0	2
complex_alt	3	3	0	0	0
shapes	2	2	0	0	0
complex	2	2	0	0	0
graphs	2	0	2	0	0
PVSioChecker	2	1	0	1	0
ASP	2	2	0	0	0
lnexp	1	1	0	0	0
metric_space	1	1	0	0	0
measure_integration	1	1	0	0	0
linear_algebra	1	1	0	0	0

The libraries with no recursive definitions have been omitted from this table

```
sqrt_newton_fast_approx(X: ℝ>0, {Y: ℝ>0 | Y*Y ≥ X}, eps: ℝ>0):
RECURSIVE {z: ℝ>0 | z*z ≥ X ∧ z*z-X ≤ eps} =
  IF Y*Y-X ≤ eps THEN Y
  ELSE
    sqrt_newton_fast_approx(X, sqrt_newton_step_fast_approx(X,Y, eps/2), eps)
  ENDIF
MEASURE IF Y*Y-X ≤ eps THEN 0 ELSE 1+log_nat((Y*Y-X)/eps,4)‘1 ENDIF
```

Fig. 8 Example of recursive declaration from group 4

root operation. This measure includes a case split, and a base-4 logarithm of a combination of three numerical parameters.

As expected, the vast majority of the termination proofs rely on simple measures and orders. The termination of about the 73% of the recursive declarations in NASALib (group 1) can be proved using the simplest kind of measures and this trend stands similarly inside almost every library. While the termination TCCs generated by the type checker on the functions from that group are automatically discharged by the built-in proof strategies proposed by default by PVS, the termination-related proof obligations generated on functions from the second group require some user guidance. The groups 3 and 4, which involve more complicated measures, account for no more than 25% of the total of recursive definitions. Nevertheless, the proofs of the TCCs generated by type checking the functions from these groups require a larger effort on the user side than for recursive declarations in the first two groups.

This classification of the recursive declarations in NASALib can also be helpful when developing heuristics to propose measures and orders to be used as part of the procedure outlined in Sect. 5. For example, the termination of all the functions in the first group can be proved using the mentioned trivial measures. The measures needed to prove termination of a recursive declaration from group 2 or 3 can be easily recognized by inspecting the way in which the arguments of the recursive call are expressed. In particular, for all the functions in the second group, the reduction on the size of the argument is obtained by the application of the `rest` function, that removes deterministically one element from a non-empty set. The group three functions contain two subgroups, the first subgroup using measures with some form of safe subtraction between arguments, and containing recursive calls in which one of the arguments is incremented. Since PVS supports the use of dependent types, sometimes the bound on the growth of the argument is given in its type, which could be recognized and utilized. The rest of the functions in group 3 involve recursive calls in which at least one inductively defined structure shrinks. In particular, this group is comprised of functions on lists. The operations that provoke a reduction in the size of the list are known and can be recognized by a simple syntactic analysis on the recursive calls, just looking for the use of such functions. This heuristic can be extended to other user-defined inductive datatypes as long as the available size-reduction functions are known by the system. These functions could be marked by the user using a specific annotation or keyword. It could even be possible to try to automatically recognize these kinds of functions by analyzing the user-defined lemmas regarding changes on the size of the structure.

It is important to note that the MWG technique allows for proof of termination using simpler measures, with less interaction. Returning to the example of the Ackermann function, the MWG technique proves termination based simply on the natural number order on each of the parameters. A user would not need to recognize that a lexicographic order on them is needed. As an example from group 3 functions, instead of specifying the measure function as the sum of the lengths of lists, it would be possible to prove termination having the length of each argument as the simple measure functions in the family of measures used by the

MWG procedure. In such cases, the “size” function for the datatype is generally easy to recognize, so could optimally be automatically chosen. While less hopeful, it is possible that the measures needed to prove termination of functions such as the ones in group 4 could be simpler than the ones currently used. But as stated above, the measures used for those functions are mainly ad-hoc complex expressions for which a general rule does not seem to exist.

In summary, the integration of the MWG technique in PVS, with the heuristics to propose measures discussed in this section, could automate the proof of termination for around the 86% of the recursive declarations in NASALib (the first three groups from the table). It is clear that the technique is particularly useful for the third group, which involve the interplay of measures on more than one parameter. But even for functions from the first two groups, (e.g., functions with a decrementing natural number parameter) the MWG technique offers a small relief of burden on the user. Assuming the technique can be fully automated, the user has no need to consider or specify the parameter that is decreasing, or identify it. While replacing these termination proofs is not a goal per-se, since those termination TCCs are already proven, it can be seen as an indicator of the usefulness of the technique in future developments.

6 Related Work, Future Work, and Conclusion

The termination of programs expressed in a language such as `PVS0` can be guaranteed by providing a measure on a well-founded relation that strictly decreases at every recursive call. This criterion can be traced back to Turing [23]. A related practical approach was further proposed by Floyd [8] and Hoare [9]. The inputs and outputs of program instructions are enriched with assertions (Floyd-Hoare first-order well-known pre- and post-conditions) so that if the pre-condition holds and the instruction is executed the post-condition must hold. To verify termination, these assertions are enriched with decreasing assertions that are built using a well-founded ordering according to some *measure* function on the inputs and outputs of the program. This approach can also be used in recursive functions, as shown by Boyer and Moore [6]. In this case, a measure is provided over the arguments of the function. The measure must strictly decrease at every possible recursive call. The conditions to effectively check if a recursive call is possible or not are statically given by the guards of branching instructions that lead to the function call.

In PVS, as in many other proof assistants, the user provides a measure function and a well-founded relation for each recursive function. The necessary conditions that guarantee termination are built during type checking. In this paper, these conditions are referred to as *termination TCCs* and the process that generates termination TCCs for PVS0 is formally verified against other termination criteria.

The functional language *foetus* checks termination of programs automatically by finding a lexicographic order on the parameters of the functions participating in the recursive-call chain [1]. The termination prover of *foetus* tries to build a well-founded structural order on the parameters of recursive calls such that the arguments in each call are smaller than the input parameters regarding this order. This technique operates on multi-graphs whose edges are labeled with *call matrices* providing information as the graphs and matrices used in this paper in several aspects. In *foetus* each node represents a function instead of a calling context, each edge represents a call, and the matrices labeling the edges relate the arguments used in each call under the same order relation. Nevertheless, the algebraic articulations of MWGs differ from those in that paper. One of the main differences is that MWG algebra is designed to detect the existence of orders that decrease in each possible cycle of execution and not an order that decreases after each function call. Checking for termination on all possible cycles of execution is the key concept coined in Lee et al. size-change principle seminal paper [13], further nicely implemented by Manolios and Vroon in ACL2 [14]. However, neither of these papers implemented the algebra of matrices as done by MWGs to build the transitive closure of matrix multiplication that expresses decrement on possible execution cycles on the calling context graphs through the notion of matrix positiveness. Another remarkable difference is that in MWGs, the matrices allow for the inclusion of arbitrary families of orderings on all parameters as opposed to orderings on a unique parameter, as is the case of call matrices, designed to build exclusively lexicographic orderings. Despite these differences, it is relevant to stress that such algebraic matrix construction is not novel. Indeed, the algebra of MWGs is close to the ones applied in graph searching algorithms that use the adjacency and weight matrix graph representations to solve reachability and minimum distance problems on graphs by specialized matrix multiplication (e.g., Floyd-Warshall algorithm [7]). Closer to work in this paper, Krauss formalizes the size-change termination principle in Isabelle/HOL [11]. He also developed a technology based on this principle and the dependency pair criterion to verify the termination of a class of recursive functions specified in Isabelle/HOL. CCGs are implemented in ACL2s by Manolios and Vroon, where they report that "[CCG] was able to prove termination automatically for over *98%* of the more than 10,000 functions in the regression suite [of ACL2s]" [14]. In his Ph.D. thesis, Vroon provides a pencil and paper proof of the correctness of the method based on CCGs [24].

The formalization presented in this paper includes proofs of equivalence among the following termination criteria: ε-termination, χ-termination, T_T, $\texttt{SCP}_<$, T_{SCP}, T_{CCG}, T_{MWG}, and Dutle's procedure. Other related formalizations that use or connect to the one presented here have been previously presented. For example, Alves Almeida and Ayala-Rincón formalized a notion of termination for term rewriting systems based on dependency pairs and showed how it could be related to the notions explained in this paper [2, 3]. The relation between size-change, rewriting and dependency pairs termination was first explored by Thiemann and Giesl in [20, 21]. Differently to Krauss et al. approach in [12] that proposes a transformation of functional programs into orthogonal rewriting systems, Alves Almeida discusses an approach that uses narrowing to transform functional programs allowing only Presburger arithmetic guards in their branching instructions into rewriting systems whose matching conditions correspond to the arithmetic guards. Also, Ferreira Ramos et al. presented a formalization of the undecidability of termination constructed over the computational model of the functional language `PVS0` [18]. More recently, Ferreira Ramos et al. extended the `PVS0` model to deal with programs that allow multiple functions [17]; they formalized computational properties such as Turing completeness, the Recursion theorem, Rice's theorem, the undecidability of the Halting problem, and the fixed-point theorem. The Matrix Weighted Graphs algebraic approach, which is an implementation of the CCG technique, was first presented in Avelar's Ph.D. along with its formalization in PVS [4]. That formalization does not include Dutle's procedure.

The authors are currently working on the implementation of proof strategies, based on computational reflection, that use the CCG/MWG technique to automate termination proofs of PVS recursive functions. As part of such effort, the team is looking for ways to automate two steps of the procedure that still require human intervention. Namely, the pruning of the graph and the selection of the family of measures to be used. The former issue can be addressed as a constraint solving problem, since the pruning step needs to decide if the combination of the path conditions of calling contexts that are adjacent in the graph are satisfiable or not. One possible solution is to query some off-the-shelf solver as an external oracle to decide if an edge can be removed. Additionally, the own PVS prover could be used to try to decide on the satisfiability of the constraints, reducing in this way the trust kernel of the approach. Regarding the measure family selection, some heuristics have been discussed in previous sections. Furthermore, techniques usually used to synthesize invariants of loops in imperative-language settings could also be adapted to propose more elaborate measure functions.

References

1. Abel, A.: foetus—Termination Checker for Simple Functional Programs. Programming Lab Report 474, LMU München (1998). http://www.cse.chalmers.se/~abela/foetus/
2. Alves Almeida, A.: On Termination by Dependency Pairs and Termination of First-Order Functional Specifications in PVS. Ph.D. thesis, Universidade de Brasília, Graduate Program in Informatics, Brasília, Distrito Federal, Brazil (2021). https://repositorio.unb.br/handle/10482/42296
3. Alves Almeida, A., Ayala-Rincón, M.: Formalizing the dependency pair criterion for innermost termination. Sci. Comput. Program. **195**, 102474 (2020). https://doi.org/10.1016/j.scico.2020.102474
4. Avelar, A.B.: Formalização da automação da terminação através de grafos com matrizes de medida. Ph.D. thesis, Universidade de Brasília, Departamento de Matemática, Brasília, Distrito Federal, Brazil (2015). https://repositorio.unb.br/handle/10482/18069. (**In Portuguese**)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. Springer-Verlag Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
6. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, Cambridge (1979)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
8. Floyd, R.W.: Assigning meanings to programs. Proc. Symp. Appl. Math. **19**, 19–32 (1967)
9. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
10. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, New York (2000)
11. Krauss, A.: Certified Size-Change Termination. In: Automated Deduction—CADE-21, 21st International Conference on Automated Deduction, Lecture Notes in Computer Science, vol. 4603, pp. 460–475. Springer (2007). https://doi.org/10.1007/978-3-540-73595-3_34
12. Krauss, A., Sternagel, C., Thiemann, R., Fuhs, C., Giesl, J.: Termination of Isabelle Functions via Termination of Rewriting. In: Interactive Theorem Proving - Second International Conference, ITP 2011, Lecture Notes in Computer Science, vol. 6898, pp. 152–167. Springer (2011). https://doi.org/10.1007/978-3-642-22863-6_13
13. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 81–92 (2001). https://doi.org/10.1145/360204.360210
14. Manolios, P., Vroon, D.: Termination Analysis with Calling Context Graphs. In: Computer Aided Verification, 18th International Conference, CAV, Lecture Notes in Computer Science, vol. 4144, pp. 401–414. Springer (2006). https://doi.org/10.1007/11817963_36
15. Muñoz, C.A., Ayala-Rincón, M., Moscato, M.M., Dutle, A.M., Narkawicz, A.J., Almeida, A.A., Avelar, A.B., M. Ferreira Ramos, T.: Formal Verification of Termination Criteria for First-Order Recursive Functions. In: L. Cohen, C. Kaliszyk (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021), Leibniz International Proceedings in Informatics (LIPIcs), vol. 193, pp. 27:1–27:17. Schloss Dagstuhl–Leibniz–Zentrum für Informatik, Dagstuhl, Germany (2021). 10.4230/LIPIcs.ITP.2021.27. https://drops.dagstuhl.de/opus/volltexte/2021/13922
16. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer (1992). https://doi.org/10.1007/3-540-55602-8_217
17. Ramos, T.M.F., Alves Almeida, A., Ayala-Rincón, M.: Formalization of the computational theory of a turing complete functional language model. J. Autom. Reason. **66**(4), 1031–1063 (2022). https://doi.org/10.1007/s10817-021-09615-x

18. Ramos, T.M.F., Muñoz, C., Ayala-Rincón, M., Moscato, M., Dutle, A., Narkawicz, A.: Formalization of the Undecidability of the Halting Problem for a Functional Language. In: Logic, Language, Information, and Computation, Lecture Notes in Computer Science, vol. 10944, pp. 196–209. Springer Berlin Heidelberg (2018). https://doi.org/10.1007/978-3-662-57669-4_11
19. Tarjan, R.: Enumeration of the elementary circuits of a directed graph. SIAM J. Comput. **2**(3), 211–216 (1973)
20. Thiemann, R., Giesl, J.: Size-Change Termination for Term Rewriting. In: Proceedings Rewriting Techniques and Applications, 14th International Conference, RTA, Lecture Notes in Computer Science, vol. 2706, pp. 264–278. Springer (2003). https://doi.org/10.1007/3-540-44881-0_19
21. Thiemann, R., Giesl, J.: The size-change principle and dependency pairs for termination of term rewriting. Appl. Algebra Eng. Commun. Comput. **16**(4), 229–270 (2005). https://doi.org/10.1007/s00200-005-0179-7
22. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **42**(1), 230–265 (1937)
23. Turing, A.: Checking a large routine. In: Report of a Conference High Speed Automatic Calculating-Machines, pp. 67–69. University Mathematical Laboratory (1949)
24. Vroon, D.: Automatically proving the termination of functional programs. Ph.D. thesis, Georgia Institute of Technology (2007)