Formal Verification of Termination Criteria for First-Order Recursive Functions

Cesar A. Muñoz
NASA Langley Research Center, Hampton, VA, USA

Mauricio Ayala-Rincón
Departments of Computer Science and Mathematics, Universidade de Brasília, Brazil

Mariano M. Moscato[a]
National Institute of Aerospace, Hampton, VA, USA

Aaron M. Dutle
NASA Langley Research Center, Hampton, VA, USA

[a] Corresponding author

Anthony J. Narkawicz
USA

Ariane Alves Almeida
Department of Computer Science, Universidade de Brasília, Brazil

Andréia B. Avelar da Silva
Faculdade de Planaltina, Universidade de Brasília, Brazil

Thiago M. Ferreira Ramos
Department of Computer Science, Universidade de Brasília, Brazil

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.

2012 ACM Subject Classification Theory of computation → Models of computation; Software and its engineering → Software verification; Computing methodologies → Theorem proving algorithms

Keywords and phrases Formal Verification, Termination, Calling Context Graph, PVS

Digital Object Identifier 10.4230/LIPIcs.ITP.2021.26

Funding *Mariano M. Moscato*: NASA/NIA Cooperative Agreement NNL09AA00A
Anthony J. Narkawicz: At NASA at time of contribution.

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) [11] is extensively used at NASA in the verification of safety-critical algorithms of autonomous unmanned systems.[1] These algorithms are typically specified as recursive functions whose computations are well-behaved, i.e., they 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 an algorithm that decides whether or not another algorithm terminates on a given input [13]. Turing's proof applies to algorithms written as Turing machines, but the proof extends to other formalisms for

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

12th International Conference on Interactive Theorem Proving (ITP 2021).
Editors: Liron Cohen and Cezary Kaliszyk; Article No. 26; pp. 26:1–26:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 Calculus of Inductive Constructions implemented in its type system [4]. Other theorem provers, such as ACL2, have incorporated syntactic conditions for checking termination of recursive functions [7]. In the Prototype Verification System (PVS), the user needs to provide a measure function over a well-founded relation that strictly decreases at every recursive call [11]. Despite the undecidability result, termination is routine, but is often 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 proper 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. [9]. The SCP principle states that if every infinite computation would give rise to an infinitely decreasing value sequence, 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 [10]. Avelar's PhD dissertation proposes an improvement on the CCG technique, based on a particular algebra on matrices [3]. The formalization reported in this paper includes all these criteria and proofs of equivalence between them. While the formalization itself has been available for some time as part of the NASA PVS Library[2], the goal of this paper is to report the main results. These results, which have been used in other works such as [2] and [12], have not been properly published before. Furthermore, this paper also presents a practical contribution: a mechanizable technique to automate (some) termination proofs of user-defined recursive functions in PVS.

For readability, this paper uses a stylized PVS notation. The development presented in this paper, including all lemmas and theorems, are formally verified in PVS and are available as part of the NASA PVS Library.[3]

2 PVS & PVS0

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.

[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`.

```
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)
```

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

▶ **Example 1.**

```
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 <
```

Example 1 provides a definiton of the Ackermann function in PVS. 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 Figure 1. Since all the 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.

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 $[\mathcal{V}al \to \mathcal{V}al]$ for an arbitrary generic type $\mathcal{V}al$. 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 $\mathcal{V}al$ and n is a natural number. Furthermore, $\texttt{cnst}(v)$ denotes a constant with value v, `vr` denotes a unique variable, `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 $[\mathcal{V}al \to \mathcal{V}al]$ and $[[\mathcal{V}al \times \mathcal{V}al] \to \mathcal{V}al]$, respectively. In the following, the collection of PVS0 expressions is referred to as $\texttt{PVS0Expr}_{\mathcal{V}al}$, where the type parameter for `PVS0Expr` is omitted when possible to lighten the notation. The PVS0 programs with values in $\mathcal{V}al$, denoted by $\texttt{PVS0}_{\mathcal{V}al}$, are 4-tuples of the form $(O_1, O_2, \bot, e)$, such that

- O_1 is a list of unary operators of type $[\mathcal{V}al \to \mathcal{V}al]$, 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 $[[\mathcal{V}al \times \mathcal{V}al] \to \mathcal{V}al]$, where $O_2(i)$ interprets the index i in applications of `op2`,
- $\bot$ is a constant of type $\mathcal{V}al$ representing the Boolean value *false* in the conditional construction `ite`, and
- e is a expression from `PVS0Expr`: the syntactic representation of the body of the program.

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 $\texttt{p} = (O_1, O_2, \bot, e)$ is a PVS0 program, the symbols $\texttt{p}_{O_1}$, $\texttt{p}_{O_2}$, $\texttt{p}_{\bot}$, 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 $\mathcal{V}al$, 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 $\texttt{ack} \equiv (O_1, O_2, \bot, e)$, where the type parameter $\mathcal{V}al$ of PVS0 is instantiated with the type of pair 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.

- $O_1(0)((m,n)) \equiv \texttt{IF}\ m = 0\ \texttt{THEN}\ \top\ \texttt{ELSE}\ \bot\ \texttt{ENDIF}$.
- $O_1(1)((m,n)) \equiv \texttt{IF}\ n = 0\ \texttt{THEN}\ \top\ \texttt{ELSE}\ \bot\ \texttt{ENDIF}$.
- $O_1(2)((m,n)) \equiv (n+1, 0)$.
- $O_1(3)((m,n)) \equiv \texttt{IF}\ m = 0\ \texttt{THEN}\ \bot\ \texttt{ELSE}\ (\max(0, m-1), 1)\ \texttt{ENDIF}$.
- $O_1(4)((m,n)) \equiv \texttt{IF}\ n = 0\ \texttt{THEN}\ \bot\ \texttt{ELSE}\ (m, \max(0, n-1))\ \texttt{ENDIF}$.
- $O_2(0)((m,n),(i,j)) \equiv \texttt{IF}\ m = 0\ \texttt{THEN}\ \bot\ \texttt{ELSE}\ (\max(0, m-1), i)\ \texttt{ENDIF}$.
- $\bot \equiv (0,0)$, and for convenience $\top \equiv (1,0)$.
- e ≡ `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.

2.1 Semantic Relation

Given a PVS0 program `p`, the semantic evaluation of a `PVS0Expr` 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 3** (Semantic Relation). *Let* $\texttt{p}$ *be a PVS0 program on a generic type* $\mathcal{V}al$, e_i *be an expression in* `PVS0Expr`, *and* v_i, v_o, v, v', v'' *be values from* $\mathcal{V}al$. *The relation* $\varepsilon(\texttt{p})(e_i, v_i, v_o)$ *holds if and only if*

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

where

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

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

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

▶ **Lemma 4.** *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 $\varepsilon(\mathtt{ack})(\mathtt{ack}_e, (m,n), (k,i))$.*

This lemma can be proved 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 5.** *Let $\mathtt{p}$ be a PVS0 program. For any `PVS0Expr` expression e_i and values $v_i, v'_o, v''_o \in \mathcal{V}al$, $\varepsilon(\mathtt{p})(e_i, v_i, v'_o)$ and $\varepsilon(\mathtt{p})(e_i, v_i, v''_o)$ implies $v'_o = v''_o$.*

PVS0 enables the encoding on 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 6** (ε-determination)**.** *A `PVS0` program $\mathtt{p}$ is said to be ε-*determined* for an input value $v_i \in \mathcal{V}al$ (denoted by $D_\varepsilon(\mathtt{p}, v_i)$) when $\exists v_o \in \mathcal{V}al : \varepsilon(\mathtt{p})(\mathtt{p}_e, v_i, v_o)$.*

2.2 Functional Semantics

The operational semantics of PVS0 can be expressed by a function $\chi : [\mathtt{PVS0} \to [\mathtt{PVS0Expr} \times \mathcal{V}al \times \mathbb{N}] \to \mathcal{V}al \uplus \{\diamond\}]$. This function returns either a value of type $\mathcal{V}al$ or a distinguished value $\diamond \notin \mathcal{V}al$. 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 7** (Semantic Function)**.** *Let $\mathtt{p}$ be a `PVS0` program, e_i a `PVS0Expr` expression, v_i a value from $\mathcal{V}al$, n a natural number, $v' = \chi(\mathtt{p})(e_1, v_i, n)$, and $v'' = \chi(\mathtt{p})(e_2, v_i, n)$.*

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

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

▶ **Theorem 8.** *For any PVS0 program p, $v_i, v_o \in \mathcal{V}al$ and $e_i \in$ PVS0Expr, $\varepsilon(p)(e_i, v_i, v_o)$ if and only if $v_o = \chi(p)(e_i, v_i, n)$, 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 9** (χ-determination)**.** *A PVS0 program p is said to be* χ-determined *for an input value $v_i \in \mathcal{V}al$ (denoted by $D_\chi(p, v_i)$) when there is an $n \in \mathbb{N}$ such that $\chi(p)(p_e, v_i, n) \neq \diamond$.*

As a corollary of Theorem 8, the notions of ε-determination and χ-determination coincide.

▶ **Theorem 10.** *For all $p \in PVS0_{\mathcal{V}al}$ and value $v_i : \mathcal{V}al$, $D_\varepsilon(p, v_i)$ if and only if $D_\chi(p, v_i)$.*

In Definition 9, there may be multiple (in fact, infinite) natural numbers n that satisfy $\chi(\mathtt{p})(\mathtt{p}_e, v_i, n) \neq \diamond$. The following definition distinguishes the minimum of those numbers.

▶ **Definition 11** (μ)**.** *Let p be a PVS0 program and v_i a value in $\mathcal{V}al$ such that $D_\chi(p, v_i)$, the minimum number of recursive calls needed to produce a result (denoted by $\mu(p, v_i)$) is formally defined as* $\min(\{n \in \mathbb{N} \mid \chi(p)(p_e, v_i, n) \neq \diamond\})$.

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 $\mathcal{V}al$. This remark is formalized by the following lemma.

▶ **Lemma 12.** *Let p be a PVS0 program and v_i a value from $\mathcal{V}al$ such that $D_\chi(p, v_i)$. For any $n \in \mathbb{N}$ such that $n \geq \mu(p, v_i)$, $\chi(p)(p_e, v_i, n) = \chi(p)(p_e, v_i, \mu(p, v_i))$.*

2.3 Semantic Termination

The notion of termination for PVS0 programs is defined using the notions of determination from Section 2.2.

▶ **Definition 13** (ε-termination and χ-termination)**.** *A PVS0 program $p \in PVS0_{\mathcal{V}al}$ is said to be* ε-terminating *(noted $T_\varepsilon(p)$) when $\forall v_i \in \mathcal{V}al : D_\varepsilon(p, v_i)$. It is said to be* χ-terminating *($T_\chi(p)$) when $\forall v_i \in \mathcal{V}al : D_\chi(p, v_i)$.*

As a corollary of Theorem 10, the notions of ε-termination and χ-termination coincide.

▶ **Theorem 14.** *For every PVS0 program p, $T_\varepsilon(p)$ if and only if $T_\chi(p)$.*

Not all PVS0 programs are terminating. For example, consider the PVS0 program $\mathtt{p}'$ with body $\mathtt{rec(vr)}$. It can be proven that $D_\varepsilon(\mathtt{p}', v_i)$ does not hold for any $v_i \in \mathcal{V}al$. 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 $\mathcal{V}al$, such as $\diamond$.

▶ **Definition 15.** *Let $PVS0_{\downarrow_\varepsilon}$ be the collection of PVS0 programs for which T_ε holds, let $p \in PVS0_{\downarrow_\varepsilon}$, and v_i be a value in $\mathcal{V}al$. The semantic function for terminating programs $\epsilon : [PVS0_{\downarrow_\varepsilon} \to \mathcal{V}al \to \mathcal{V}al]$ is defined in the following way.*

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

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

▶ **Theorem 16.** *For all terminating PVS0 program* $\mathtt{p}$*, i.e.,* $T_\varepsilon(\mathtt{p})$ *holds, and values* $v_i, v_o \in \mathcal{V}al$*,* $\varepsilon(\mathtt{p})(\mathtt{p}_e, v_i, v_o)$ *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 $\mathcal{V}al$. 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 Section 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, Figure 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 17** (Valid Path)**.** *Let* $\mathtt{p}$ *be a PVS0 program, a finite sequence of natural numbers* p *is a* Valid Path *of* $\mathtt{p}$ *if* p *determines a path in the labeled syntax tree of* $\mathtt{p}$ *from any node* e *to the root of the tree. In that case,* p *is said to* reach e *in* $\mathtt{p}$*.*

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 exercise the particular subexpression e when used as inputs for the evaluation of `p`. 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

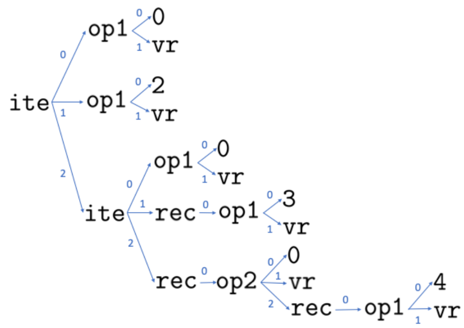


Figure 2 Abstract syntax tree of the Ackermann function from Example 2.

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 18** (Polarized Expression). *Given a $\mathtt{PVS0Expr}$ expression e, the* polarized version *of e is a pair $[\mathtt{PVS0Expr}\times\ \{0,1\}]$ 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 $\mathtt{p}$, 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 19** (Path Conditions). *Let p be a valid path of a PVS0 program $\mathtt{p}$ and e the subexpression of $\mathtt{p}_e$ reached by p. The list of polarized guard expressions of $\mathtt{p}$ that are needed to be valid in order for the evaluation of $\mathtt{p}$ to involve the expression e is called the* list of path conditions *of p.*

▶ **Definition 20** (Calling Context). *A* calling context *of a program $\mathtt{p}$ is a tuple $(\mathtt{rec}(e'),p,\mathbf{c})$ containing: a path p, which is valid in $\mathtt{p}$, 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 $\mathtt{p}$ 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 21.** The calling contexts for the `ack` function from Example 2 are:

- $(\mathtt{rec(op1(3,vr))}, \langle 1,2\rangle, \langle \neg\mathtt{op1(0,vr)}, \mathtt{op1(1,vr)}\rangle)$,
- $(\mathtt{rec(op2(0,vr,rec(op1(4,vr))))}, \langle 2,2\rangle, \langle \neg\mathtt{op1(0,vr)}, \neg\mathtt{op1(1,vr)}\rangle)$, and

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

An input value v_i is said to *exercise* a calling context $\mathbf{cc} = (e, p, \mathbf{c})$ in a program $\mathtt{p}$ when $\varepsilon_\pm(\mathtt{p})(c, v_i)$ holds. A program $\mathtt{p}$ is TCC-terminating if for each calling context $\mathbf{cc}$ in $\mathtt{p}$ 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 is considered smaller than another one if the former is closer to the bottom induced by a well-founded relation than the latter.

▶ **Definition 22** (TCC-termination). *A PVS0 program $\mathtt{p}$ is said to be* TCC-terminating*, or* Turing-terminating*, on a measuring type M if there exist a function $m : [\mathcal{V}al \to M]$ and a well-founded relation $<_M$ on M such that for all calling context $\mathbf{cc} = (\mathtt{rec}(e), p, \mathbf{c})$ among the calling contexts of $\mathtt{p}$, for all $v_i, v_o \in \mathcal{V}al$, if $\varepsilon_\pm(\mathtt{p})(\mathbf{c}, v_i)$ and $\varepsilon(\mathtt{p})(e, v_i, v_o)$ hold, then $m(v_o) <_M m(v_i)$.*

The notion of TCC-termination on a program $\mathtt{p}$ is denoted by the predicate $T_T^{[M,<_M,m]}(\mathtt{p})$, 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 25 below. A key construction used in the proof of Theorem 25 is the function Ω, defined as follows.

▶ **Definition 23** (Ω). *Let $<_{p,m}$ be a binary relation on $\mathcal{V}al$ defined as $v_1 <_{p,m} v_2$ if and only if $m(v_1) <_M m(v_2)$ and the evaluation of $\mathtt{p}$ with v_2 as input reaches a recursive call $\mathtt{rec}(e)$ such that $\varepsilon(\mathtt{p})(e, v_2, v_1)$ holds. Then, $\Omega_{p,m}(v) \equiv \min(\{i : \mathbb{N}^+ \mid \forall\, v' \in \mathcal{V}al : \neg(v' <^i_{p,m} v)\})$ where $v' <^i_{p,m} v$ denotes a chain of $i+1$ values related by $<_{p,m}$ with endpoints in v' and v.*

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 24.** *Let $\mathtt{p}$ be a TCC-terminating 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 $v \in \mathcal{V}al$, $\mu(\mathtt{p}, v) \le \Omega_{\mathtt{p},m}(v)$.*

▶ **Theorem 25.** *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 14, $D_\chi(\mathtt{p}, v)$ holds as well. Following the definition of χ and the determinism of ε (Lemma 5), 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 $v \in \mathcal{V}al$ and any subexpression e of $\mathtt{p}$, there exists a natural number $n \le \Omega_{\mathtt{p},m}(v)$ such that $\chi(\mathtt{p})(e, v_i, n) \ne \diamond$, which assures $T_\varepsilon(\mathtt{p})$ by Theorem 14. 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 25 can be used as a practical tool to prove ε-termination of PVS0 programs, as illustrated by the following lemma.

▶ **Lemma 26.** *The PVS0 program $\mathtt{ack}$ from Example 2 is ε-terminating, i.e., $T_\varepsilon(\mathtt{ack})$ holds.*

Proof. In order to use the Theorem 25, 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

$T_T^{[M,<_M,m]}$(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 $T_T^{[[\mathbb{N}\times\mathbb{N}],<_{lex},id]}$(ack) holds, it suffices to check that for every input pair v_i, leading to any of the recursive-call subexpressions rec(e) in ack, v_i is such that for every pair v_o satisfying ε(ack)(e, v_i, v_o), $v_o <_{lex} v_i$.

There are only three recursive calls in ack (see Example 2), 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 `rec(op1(3,vr))` 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 op1$(3,\cdot)$, for every v_o such that ε(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 ε(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, $T_T^{[[\mathbb{N}\times\mathbb{N}],<_{lex},id]}$(ack) holds and, by Theorem 25, T_ε(ack) holds as well. ◀

The inequalities of the form $v_o <_{lex} v_i$ that are proved in Lemma 26 correspond to the actual termination correctness conditions generated by the PVS type checker for the function `ackermann` defined in Example 1.

4 Calling Context Graphs

The Size Change Principle (SCP) 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" [9]. Calling Context Graphs is a technique that implements the SCP [10].

▶ **Definition 27** (Valid Trace). *Given* $\mathtt{p} \in$ *PVS0, an infinite sequence* $\mathbf{cc} = \langle \mathtt{rec}(e_i), p_i, \mathbf{c}_i\rangle_{i\in\mathbb{N}}$ *of calling contexts of* p*, and an infinite sequence of values* $\mathbf{v}$ *from* $\mathcal{V}al$*,* $\mathbf{cc}$ *and* $\mathbf{v}$ *are said to form a* valid trace of calls *if the following predicate* τ *holds.*[4]

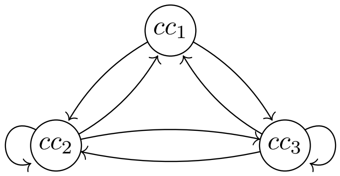
$$\tau_{\mathtt{p}}(\mathbf{cc}, \mathbf{v}) \equiv \forall (i : nat) : (\varepsilon_\pm(\mathtt{p})(\mathbf{c}_i, \mathbf{v}_i) \wedge \varepsilon(\mathtt{p})(e_i, \mathbf{v}_i, \mathbf{v}_{i+1})).$$

▶ **Definition 28** (SCP-Termination). *A PVS0 program* p *is said to be* SCP-terminating*, denoted by* $T_{SCP}(\mathtt{p})$*, if there are no infinite sequence* $\mathbf{cc}$ *of calling contexts of* p *and no infinite sequence* $\mathbf{v}$ *of values in* $\mathcal{V}al$ *such that* $\tau(\mathbf{cc}, \mathbf{v})$ *holds.*

▶ **Theorem 29.** *For all* $\mathtt{p} \in$ *PVS0,* $T_\varepsilon(\mathtt{p})$ *if and only if* $T_{SCP}(\mathtt{p})$*.*

Proof. By Theorem 25 it is enough to prove that $T_T(\mathtt{p})$ and $T_{SCP}(\mathtt{p})$ are equivalent. Proving $T_{SCP}(\mathtt{p})$ given $T_T(\mathtt{p})$ is straightforward. To prove the other direction, it is necessary to use $\Omega_{\mathtt{p},m}$. Since one has $T_{SCP}(\mathtt{p})$, it is possible to provide a relation between parameters and arguments of recursive calls and prove that it is well-founded. Similarly to the proof of Theorem 25, the closure of this relation is then used to parametrize the function $\Omega_{\mathtt{p},m}$, which provides the height of the tree of evaluation of recursive calls as the needed measure. ◀

[4] Since $\varepsilon_\pm$ can be straightforwardly extended to lists of polarized expressions, the same symbol is used for both versions along the text.



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

Figure 3 A possible CCG for the Ackermann function.

▶ **Definition 30.** *Let $<$ be a well-founded relation over $\mathcal{V}al$, $\mathtt{SCP}_<(\mathtt{p})$ holds if for all infinite sequence $\mathbf{cc}$ of calling contexts of $\mathtt{p}$ and for all infinite sequence $\mathbf{v}$ of values in $\mathcal{V}al$ 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 31.** *For all $\mathtt{p} \in \mathtt{PVSO}_{\mathcal{V}al}$, $T_{SCP}(\mathtt{p})$ if and only if $\mathtt{SCP}_<(\mathtt{p})$ for a well-founded relation $<$ over $\mathcal{V}al$.*

The proof of Theorem 31 uses the fact that every well-founded order provides a non-infinite decreasing sequence of elements.

▶ **Definition 32.** *A* Calling Context Graph *of a PVS0 program $\mathtt{p}$ ($\mathtt{p} \in \mathtt{PVSO}_{\mathcal{V}al}$) 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*

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

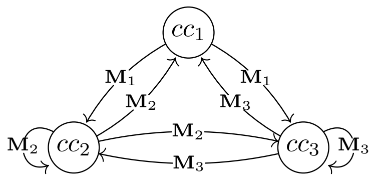
then the edge $\langle(\mathtt{rec}(e_a), P_a, C_a), (\mathtt{rec}(e_b), P_b, C_b)\rangle \in E$.

The condition on the edges admits any fully connected graph of calling contexts to be considered a CCG. For the sake of exemplification, another possible CCG for the Ackermann function as defined in the Example 1 is depicted in the Figure 3, where the calling contexts from Example 21 are abbreviated to improve readability. The lack of the loop on cc_1 does not prevent the graph to be considered a CCG because there exist no tuples $(a, b), (c, d) \in [\mathbb{N} \times \mathbb{N}]$ such that $\varepsilon_\pm(\mathtt{ack})(C_{cc_1}, (a, b)) \wedge \varepsilon(\mathtt{ack})(e_{cc_1}, (a, b), (c, d)) \wedge \varepsilon_\pm(\mathtt{ack})(C_{cc_2}, (c, d))$, since this formula can be expanded to $(a \neq 0 \wedge b = 0) \wedge (c = a - 1 \wedge d = 1) \wedge (c \neq 0 \wedge d = 0)$.

The following standard notions from Graph Theory will be used in the definitions below. A *walk* of $G_\mathtt{p}$ 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 length of a 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 33.** *Let $\mathcal{M}$ be a* family *of N measures $\mu_k : \mathcal{V}al \to M$, 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 $1 \leq k_j \leq N$ representing measure μ_{k_j}, such that for all $1 \leq j < n$, $v, v' \in \mathcal{V}al$, $\varepsilon_\pm(\mathtt{p})(C_j, v) \wedge \varepsilon(\mathtt{p})(e_j, v, v')$ implies $\mu_{k_j}(v) \rhd_j \mu_{k_{j+1}}(v')$, where $cc_{i_j} = (\mathtt{rec}(e_j), P_j, C_j)$ and $\rhd_j \in \{>, \geq\}$. A measure combination is* descending *if at least one $\rhd_j$ is $>$.*

▶ **Definition 34.** *Let $G_\mathtt{p}$ be a CCG of a PVS0 program $\mathtt{p} \in \mathtt{PVSO}_{\mathcal{V}al}$ and let $\mathcal{M}$ be a family of measures for a well-founded relation $<$ over a type M. The graph $G_\mathtt{p}$ is said to be* CCG terminating *(denoted by $T_{CCG}(G_\mathtt{p})$) if for all circuits $cc_{i_1}, \ldots, cc_{i_n}$ in $\mathbf{Walk}_{G_p}$ there is a* descending *measure combination $k_1, \ldots, k_n$, with $k_1 = k_n$.*



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

Figure 4 A MWG for the p program for the Ackermann function, where the family of measures $\mathcal{M}$ is composed by $\mu_1(m,n) = m$ and $\mu_2(m,n) = n$.

▶ **Theorem 35.** *For all* $\mathtt{p} \in \mathit{PVS0}_{\mathcal{V}al}$*,* $T_{SCP}(\mathtt{p})$ *if and only if* $T_{CCG}(G_{\mathtt{p}})$ *for some CCG* $G_{\mathtt{p}}$ *of* $\mathtt{p}$ *and some family of measures* $\mathcal{M}$*.*

Since the number of circuits in a CCG is potentially infinite, CCG termination does not directly provide an effective procedure to check termination. Even though the number of cycles in a graph is indeed finite, it is not enough to check for decreasing measure combinations in cycles (see [3] for details).

5 Matrix-Weighted Graphs

Matrix-Weighted Graphs is a technique to check for descending measure combinations in a CCG using an algebra over matrices [3]. Let $\mathcal{M}$ be a family of N measures, every edge in the CCG is labeled with a matrix of dimension $N \times N$ and values in $\{-1, 0, 1\}$. The type of these matrices will be denoted by $\mathbb{M}^N_{\mathbf{3}}$.

▶ **Definition 36** (Matrix Weighted Graph)**.** *Let* $\mathtt{p}$ *be a* PVS0 *program in* $\mathit{PVS0}_{\mathcal{V}al}$ *and* $\mathcal{M}$ *be a family of* N *measures* $\{\mu_i\}_{i=1}^N$*. A* matrix-weighted graph $W_{\mathtt{p}}^{\mathcal{M}}$ *of* $\mathtt{p}$ *is a CCG* $G_{\mathtt{p}} = (V, E)$ *of* $\mathtt{p}$ *whose edges are* correctly *labeled by matrices in* $\mathbb{M}^N_{\mathbf{3}}$*.*

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* $\mathbf{M}_{ab}(i,j) = 1$*, for all* $v_a, v_b \in \mathcal{V}al$*,* $\varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b)$ *implies* $\mu_i(v_a) > \mu_j(v_b)$*.*
- *if* $\mathbf{M}_{ab}(i,j) = 0$*, for all* $v_a, v_b \in \mathcal{V}al$*,* $\varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b)$ *implies* $\mu_i(v_a) \geq \mu_j(v_b)$*.*

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

The Figure 4 depicts a possible MWG for the p program implementing the Ackermann function.

The algebra of matrices used to define the notion of MWG termination is given by the following operations. Multiplication of matrices with values in $\{-1, 0, 1\}$ is defined as usual, where addition and multiplication of such values is defined below. Let $x, y \in \{-1, 0, 1\}$,

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

▶ **Definition 37** (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 38.** Continuing the example in Figure 4, 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 39.** *Let W_p be an MWG for a PVS0 program p and $\mathbf{w} = cc_1, \cdots, cc_n$ be a walk of W_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, 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 40** (Matrix-Weighted Graph Termination)**.** *Let p a PVS0 program and let W_p be a MWG of p. The graph W_p is said to be* MWG terminating *(denoted by $T_{MWG}(W_{\mathrm{p}})$) when for every circuit $\mathbf{w_i}$ of W_p, $w(\mathbf{w_i})$ is positive.*

The equivalence between the notions of termination for CCG and MWG is stated by Theorem 41 below.

▶ **Theorem 41.** *Let $\mathcal{M}$ be a family of N measures for a well-founded relation $<$ over a type M. For all $p \in PVS0_{\mathcal{V}al}$, $T_{CCG}(C_p^{\mathcal{M}})$ for some CCG $C_p^{\mathcal{M}}$ if and only if $T_{MWG}(W_p^{\mathcal{M}})$ for some MWG $W_p^{\mathcal{M}}$.*

Proof. This theorem follows from the fact that circuits in W_{p}, built from G_{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_{p}. ◀

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

Proof. In order to prove $T_{MWG}(W_{\mathrm{p}})$, it is necessary to show that every circuit of W_{p} has positive weight. For every circuit $\mathbf{w} = cc_1, \cdots, cc_n$ of W_{p}, if $n \leq |W_{\mathrm{p}}| \cdot 3^{N^2} + 1$, 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 $|\mathbf{w}'| \leq |W_{\mathrm{p}}| \cdot 3^{N^2} + 1$. Hence, by hypothesis, $w(\mathbf{w})'$ is positive and then $w(\mathbf{w})$ is positive too.

The existence of the circuit $\mathbf{w}'$ can be shown by constructing a sequence of pairs $\langle (cc_i, w(cc_1, \cdots, cc_i)) \rangle_{i=1}^{n}$, where for each $1 \leq i \leq n$, 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. By a simple counting argument, it can be seen that there cannot exist more than $|W_{\mathrm{p}}| \cdot 3^{N^2}$ of these pairs. Since $n > |W_{\mathrm{p}}| \cdot 3^{N^2} + 1$, 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 $i \neq j$. Without loss of generality, it can be assumed that $i < j$. Then, the walk $\mathbf{w}'' = cc_1, \cdots, cc_{i-1}, cc_j, cc_{j+1}, \cdots, cc_n$ is a circuit, since $cc_i = cc_j$ and $cc_1 = cc_n$, and it is shorter than $\mathbf{w}$. To calculate the

```
terminating?(W_p: MWG): bool =
  LET f_1 ← expandWeightLists(W_p, λ(v : V_{W_p}) : null)
  IN terminatingAt?(W_p, 1, f_1)

terminatingAt?(W_p: MWG, i : ℕ, f_i : [V_{W_p} → list[𝕄_3^N]]): bool =
 i ≥ |W_p| · 3^{N^2} + 1 OR
 LET f_{i+1} ← expandWeightLists(W_p, f_i) IN
 IF ∃ (cc ∈ V_{W_p}, M ∈ f_{i+1}(cc)) : ¬ positive?(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} → list[𝕄_3^N]]): [V_{W_p} → list[𝕄_3^N]] =
 λ(v : V_{W_p}): map(expandPartialWeight(f_i), allCyclesAt(W_p, v))

expandPartialWeight(f_i : [V_{W_p} → list[𝕄_3^N]]): [Walk_{W_p} → list[𝕄_3^N]] =
 λ(w : Walk_{W_p}):
   LET l ← cons(id_×, f_i(w[0]))
   IN IF |w| = 1 THEN l
      ELSE LET l_1 ← map(λ (M: 𝕄_3^N): M * w(w[0..1]))(l),
               l_2 ← expandPartialWeight(w[1 .. |w| − 1], f_i)
           IN pairwiseMultiplication(l_1, l_2) ENDIF
```

Figure 5 Dutle's procedure to check termination on matrix-weighted graphs.

length of $\mathbf{w}''$, first it should be noted that, by Lemma 39, $w(cc_1, \cdots, cc_i, cc_{j+1}, \cdots, cc_n) = w(cc_1, \cdots, cc_{i-1}, cc_j) \times w(cc_j, cc_{j+1}, \cdots, cc_n)$. Since $cc_i = cc_j$ and $w(cc_1, \cdots, cc_i) = w(cc_1, \cdots, cc_j)$, $w(\mathbf{w}'') = w(cc_1, \cdots, cc_j) \times w(cc_j, cc_{j+1}, \cdots, cc_n)$, which by Lemma 39 again is equal to $w(\mathbf{w})$.

If the length of $\mathbf{w}''$ is at most $|W_\mathtt{p}| \cdot 3^{N^2} + 1$, it can be taken to be $\mathbf{w}'$. Otherwise, the same procedure can be repeated to shorten the circuit even further. Since this procedure removes at least one vertex each time, eventually a circuit shorter than $|W_\mathtt{p}| \cdot 3^{N^2} + 1$ and with the same weight than $\mathbf{w}$ will be obtained. ◀

Lemma 42 allows for the definition of a procedure to check termination on a matrix-weighted graph. This procedure is referred to as Dutle's procedure. Given a MWG $W_\mathtt{p}^\mathcal{M} = (V, E)$ on a family of N measures $\mathcal{M}$ 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 $1 \leq i \leq |W_\mathtt{p}| \cdot 3^{N^2} + 1$. These functions are such that for each vertex $cc \in V$ and every circuit $\mathbf{w}$ in $W_\mathtt{p}^\mathcal{M}$ starting at cc and $|\mathbf{w}| <= i$, there is a weight $\mathbf{M} \in f_i(cc)$ for which $\mathbf{M} \leq w(\mathbf{w})$. If for some i there is 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 $W_\mathtt{p}^\mathcal{M}$ is not terminating, since there is a circuit whose weight is not positive. On the contrary, if the algorithm 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, $W_\mathtt{p}^\mathcal{M}$ can be safely declared as terminating thanks to Lemma 42.

Figure 5 depicts a pseudocode for Dutle's procedure. The function **terminatingAt**? implements the rough idea described in the previous paragraph. The auxiliary function **expandWeightLists** computes f_{i+1} given its predecessor f_i. Hence, for instance, f_1 contains lower bounds for the weight of each cycle in the graph $W_\mathtt{p}$. Starting from there, in every recursive call to **terminatingAt**?, for each vertex cc in $W_\mathtt{p}$, $f_{i+1}(cc)$ grows with respect to $f_i(cc)$ by incorporating lower bounds for the circuits passing through cc that are longer that the ones considered in $f_i(cc)$ by a complete cycle each. Then, f_i provides information about a lower bound on each walk of length at most i as previously stated, but it also contains information about longer circuits. Hence, a guard that checks saturation of such functions $(f_{i+1} = f_i)$ is also included to prematurely end the recursion if possible.

In the pseudocode, $\mathbf{cons}(x, l)$ denotes the list constructed from the element x and the list l, **null** denotes the empty list, and $\mathbf{map}(f, l)$ is used to denote the list formed by the application of the function f to each element in l. Furthermore, $\mathbf{positive}?(\mathbf{M})$ checks if a matrix $\mathbf{M}$ is positive in the sense of Definition 37, $\mathbf{allCyclesAt}(G, v)$ returns the list of all the cycles in the graph G passing through node v (if any), $\mathbf{id}_\times$ denotes the matrix weight that acts as multiplicative identity, and $\mathbf{pairwiseMultiplication}(l_1, l_2)$ is the funtion that 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.

Dutle's Procedure is a sound and complete procedure to decide positive weight of all circuits in a matrix-weighted graph and hence to check termination on MWG. This procedure has been formally verified in PVS as part of this work. The performance of the procedure can be improved in both execution time and used storage space. For example, the function **expandWeightLists** keeps enlarging the lists on 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 $\mathbf{M}' \leq \mathbf{M}$.

The notion of Matrix Weighted Termination can be used to define a procedure to automatically prove termination of certain recursive functions in PVS. Such a procedure consist 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).
   - 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 $\vdash \forall (v_a, v_b : \mathcal{V}al) : \varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b) \Rightarrow \neg\, \varepsilon_{\pm}(\mathtt{p})(C_b, v_b)$ can be discharged.
   - In order to select measures to form the family $\mathcal{M}$, 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 $\vdash \forall (v_a, v_b : \mathcal{V}al) : \varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b) \Rightarrow \mu_i(v_a) > \mu_j(v_b)$ can be proved, set $M_a(i, j)$ to 1.
   - If $\vdash \forall (v_a, v_b : \mathcal{V}al) : \varepsilon_{\pm}(\mathtt{p})(C_a, v_a) \wedge \varepsilon(\mathtt{p})(e_a, v_a, v_b) \Rightarrow \mu_i(v_a) \geq \mu_j(v_b)$ can be proved, set $M_a(i, j)$ to 0.
4. Use Dutle's procedure to check termination on the MWG.

6 Conclusion, Related and Future Work

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 [14]. A related practical approach was further proposed by Floyd [6]. 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 [5]. 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 the case of 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 Agda tries to automatically check termination of programs by finding a lexicographic order on the parameters of the functions participating in the recursive-call chain [1]. This technique operates on multi-graphs whose edges are labeled with matrices, but they differ from the graphs and matrices used in this paper in several aspects. In that paper, 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, instead of different measures as in the technique presented in this paper. Closer to the work in this paper, Krauss formalizes the size-change termination principle in Isabelle/HOL [8]. 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 automatically prove termination for over *98%* of the more than 10,000 functions in the regression suite [of ACL2s]" [10]. In his PhD thesis, Vroon provides a pencil and paper proof of the correctness of his method based on CCGs [15].

The formalization presented in this paper includes proofs of equivalence among several termination criteria. Other related formalizations that use or connect to the one presented in this paper 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 can be related to the notions explained in this paper [2]. Also, Ferreira Ramos et. al. have presented a proof of termination undecidability constructed on the model language `PVS0` [12]. The Matrix Weighted Graphs algebraic approach, which is an implementation of the CCG technique, was first presented in Avelar's PhD along with its formalization in PVS [3]. 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.

7 Bibliography

References

1 Andreas Abel. *foetus* – Termination checker for simple functional programs. Programming Lab Report 474, LMU München, 1998. URL: `http://www.cse.chalmers.se/~abela/foetus/`.

2 Ariane Alves Almeida and Mauricio Ayala-Rincón. Formalizing the dependency pair criterion for innermost termination. *Sci. Comput. Program.*, 195:102474, 2020. `doi:10.1016/j.scico.2020.102474`.

3 Andréia B. Avelar. *Formalização da automação da terminação através de grafos com matrizes de medida*. PhD thesis, Universidade de Brasília, Departamento de Matemática, Brasília, Distrito Federal, Brasil, 2015. In Portuguese.

4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag Berlin Heidelberg, 2004. `doi:10.1007/978-3-662-07964-5`.

5 Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

6 Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

7 Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

8 Alexander Krauss. Certified size-change termination. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 460–475, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

9 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2001. `doi:10.1145/360204.360210`.

10 Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 401–414, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

11 Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Proceedings of CADE 1992*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

12 Thiago Mendonça Ferreira Ramos, César Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, and Anthony Narkawicz. Formalization of the undecidability of the halting problem for a functional language. In Lawrence S. Moss, Ruy de Queiroz, and Maricarmen Martinez, editors, *Logic, Language, Information, and Computation*, pages 196–209, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

13 Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42(1):230–265, 1937.

14 Alan Turing. Checking a large routine. In *Report of a Conference High Speed Automatic Calculating-Machines*, pages 67–69. University Mathematical Laboratory, 1949.

15 Daron Vroon. *Automatically Proving the Termination of Functional Programs*. PhD thesis, Georgia Institute of Technology, 2007.