Formalization of the Undecidability of the Halting Problem for a Functional Language

Thiago Mendonça Ferreira Ramos[1], César Muñoz[2], Mauricio Ayala-Rincón[1],
Mariano Moscato[3], Aaron Dutle[2], and Anthony Narkawicz[2]

[1] University of Brasília, Brasília, Brazil
tramos@aluno.unb.br, ayala@unb.br
[2] NASA Langley Research Center, Hampton, VA, USA
{cesar.a.munoz,aaron.m.dutle,anthony.narkawicz}@nasa.gov
[3] National Institute of Aerospace, Hampton, VA, USA
mariano.moscato@nianet.org

Abstract. This paper presents a formalization of the proof of the undecidability of the halting problem for a functional programming language. The computational model consists of a simple first-order functional language called PVS0 whose operational semantics is specified in the Prototype Verification System (PVS). The formalization is part of a termination analysis library in PVS that includes the specification and equivalence proofs of several notions of termination. The proof of the undecidability of the halting problem required classical constructions such as mappings between naturals and PVS0 programs and inputs. These constructs are used to disprove the existence of a PVS0 program that decides termination of other programs, which gives rise to a contradiction.

1 Introduction

In computer science, program termination is the quintessential example of a property that is undecidable, a fact that is well-known as the *undecidability of the halting problem* [12]. This undecidability implies that it is not possible to build a compiler that would verify whether a program terminates for any given input. Despite this undecidability, it is possible to construct algorithms that partially decide termination, i.e., they correctly answer whether an input program"terminates or not", but may also answer "do not know". Termination analysis of programs is an active area of research. Indeed, substantial progress in this area is regularly presented in meetings such as the International Workshop on Termination and the Annual International Termination Competition.

To formally verify correctness of termination analysis algorithms, it is often necessary to specify and prove equivalence among multiple notions of termination. Given a formal model of computation, one natural notion of termination is specified as *for all inputs there exists an output provided under the operational semantics of the model.* Another notion of termination could be specified considering whether or not the depth of the expansion tree of computation steps for all inputs is finite. These two notions rely on the semantics of the computational model. A more syntactic approach, attributed to Turing [13], is to verify that the actual arguments decrease in any repeating control structure, e.g., recursion, unbounded loop, fix-point, etc., of the program according to some well-founded relation. This notion is used in the majority of proof assistants, where the user must provide the well-founded relation.

The main contribution of this work is the formalization in the Prototype Verification System (PVS) [10] of the theorem of undecidability of the halting problem for a model of computation given by a functional language called PVS0. The formal development includes the definition of PVS0, its operational semantics, and the specification and proof of several concepts used in termination analysis of PVS0 programs. For the undecidability proof of the halting problem, only the semantic notions of termination are used. Turing termination for the language PVS0 is also discussed to show how semantic and syntactic termination criteria are related. The formalization is available as part of the NASA PVS Library under the directory PVS0.[4] All lemmas and theorems presented in this paper were formalized and verified in PVS.

2 Semantic Termination

The PVS0 language is a simple functional language whose expressions are described by the following grammar.

$$\mathit{expr} ::= \texttt{cnst} \mid \texttt{vr} \mid \mathit{op1}(\mathit{expr}) \mid \mathit{op2}(\mathit{expr}, \mathit{expr}) \mid \mathit{rec}(\mathit{expr}) \mid \texttt{ite}(\mathit{expr}, \mathit{expr}, \mathit{expr})$$

The grammar above is specified in PVS through the abstract data type:

```
PVS0Expr[T:TYPE+] : DATATYPE
BEGIN
  cnst(get_val:T) : cnst?
  vr : vr?
  op1(get_op:nat,get_arg:PVS0Expr) : op1?
  op2(get_op:nat,get_arg1,get_arg2:PVS0Expr) : op2?
  rec(get_arg:PVS0Expr) : rec?
  ite(get_cond,get_if,get_else:PVS0Expr) : ite?
END PVS0Expr
```

A PVS specification of an abstract data type includes the constructors, e.g., `ite`, the accessors, e.g., `get_cond`, and the recognizers, e.g., `ite?`. In this data type, `T` is a parametric nonempty type that represents the type of values that serve as inputs and outputs of PVS0 programs. Furthermore, `cnst` is the constructor of constant values of type `T`, `vr` is the unique variable constructor, `op1` and `op2` are constructors of unary and binary operators, respectively, `rec` is the constructor of recursion, and `ite` is the constructor of conditional "if-then-else" expressions.

[4] https://github.com/nasa/pvslib.

The first parameter of the constructors `op1` and `op2` is an index representing built-in unary and binary operators, respectively.

The uninterpreted type `T` and uninterpreted unary and binary operators enable the encoding of arbitrary first-order PVS functions as programs in PVS0. Indeed, the operational semantics of `PVS0Expr` is given in terms of a non-empty set *Val*, which interprets the type `T`. The type $\texttt{PVS0}[\mathit{Val}]$ of PVS0 programs with values in *Val* consists of all 4-tuples of the form $(O_1, O_2, \bot, \mathit{expr})$, such that

- O_1 is a list of PVS functions of type $\mathit{Val} \to \mathit{Val}$, where $O_1(i)$, i.e., the i-th element of the list O_1, interprets the index i in the constructor `op1`,
- O_2 is a list of PVS functions of type $\mathit{Val} \times \mathit{Val} \to \mathit{Val}$, where $O_2(i)$, i.e., the i-th element of the list O_2, interprets the index i in the constructor `op2`,
- $\bot$ is a constant of type *Val* representing the Boolean value false in the conditional construction `ite`, and
- *expr* is a $\texttt{PVS0Expr}[\mathit{Val}]$, which is the syntactic representation of the program itself.

Henceforth, $|O_1|$ and $|O_2|$ represent the length of the lists O_1 and O_2, respectively. The choice of lists of functions for interpreting unary and binary operators helps in the enumeration of PVS0 programs, which is necessary in the undecidability proof.

Given a program $(O_1, O_2, \bot, e_f)$ of type $\texttt{PVS0}[\mathit{Val}]$, the semantic evaluation of an expression e of type $\texttt{PVS0Expr}[\mathit{Val}]$ is given by the curried inductive relation ε of type $\texttt{PVS0}[\mathit{Val}] \to (\texttt{PVS0Expr}[\mathit{Val}] \times \mathit{Val} \times \mathit{Val}) \to \texttt{bool}$ defined as follows.

Intuitively, the relation $\varepsilon(O_1, O_2, \bot, e_f)(e, v_i, v_o)$ defined below holds when given a program $(O_1, O_2, \bot, e_f)$ the evaluation of the expression e on the input value v_i is the value v_o.

$$
\begin{array}{rcl}
\varepsilon(O_1, O_2, \bot, e_f)(e, v_i, v_o) & := & \texttt{CASES}\ e\ \texttt{OF} \\
\texttt{cnst}(v) & : & v_o = v; \\
\texttt{vr} & : & v_o = v_i; \\
\texttt{op1}(j, e_1) & : & j < |O_1| \ \wedge \exists\, v' \in \mathit{Val} : \\
 & & \varepsilon(O_1, O_2, \bot, e_f)(e_1, v_i, v') \wedge v_o = O_1(j)(v'); \\
\texttt{op2}(j, e_1, e_2) & : & j < |O_2| \ \wedge \exists\, v', v'' \in \mathit{Val} : \\
 & & \varepsilon(O_1, O_2, \bot, e_f)(e_1, v_i, v') \ \wedge \\
 & & \varepsilon(O_1, O_2, \bot, e_f)(e_2, v_i, v'') \ \wedge \\
 & & v_o = O_2(j)(v', v''); \\
\texttt{rec}(e_1) & : & \exists\, v' \in \mathcal{V}al : \varepsilon(O_1, O_2, \bot, e_f)(e_1, v_i, v') \ \wedge \\
 & & \varepsilon(O_1, O_2, \bot, e_f)(e_f, v', v_o) \\
\texttt{ite}(e_1, e_2, e_3) & : & \exists\, v' : \varepsilon(O_1, O_2, \bot, e_f)(e_1, v_i, v') \ \wedge \\
 & & \texttt{IF}\ v' \neq \bot\ \texttt{THEN}\ \varepsilon(O_1, O_2, \bot, e_f)(e_2, v_i, v_o) \\
 & & \texttt{ELSE}\ \varepsilon(O_1, O_2, \bot, e_f)(e_3, v_i, v_o).
\end{array}
$$

In the definition of ε, the parameters e_f and e are needed since the evaluation of a program $(O_1, O_2, \bot, e_f)$ leads to evaluation of sub expressions e of e_f and, when a recursive call is evaluated, the whole expression e_f should be considered again (see the recursive case $\texttt{rec}(e_1)$ above).

For example, consider below a PVS0 program that computes the Ackermann function.

Example 1. Let *Val* be the set $\mathbb{N} \times \mathbb{N}$ of pairs of natural numbers, $\top = (1, 0)$, $\bot = (0, 0)$, and a be the PVS0 program $(O_1, O_2, \bot, e_a)$, where

$$\begin{aligned}
O_1(0)(m, n) &:= \texttt{IF } m = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot,\\
O_1(1)(m, n) &:= \texttt{IF } n = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot,\\
O_1(2)(m, n) &:= (n + 1, 0),\\
O_1(3)(m, n) &:= \texttt{IF } m > 0 \texttt{ THEN } (m - 1, 1) \texttt{ ELSE } \bot,\\
O_1(4)(m, n) &:= \texttt{IF } n > 0 \texttt{ THEN } (m, n - 1) \texttt{ ELSE } \bot,\\
O_2(0)((m, n), (i, j)) &:= \texttt{IF } m > 0 \texttt{ THEN } (m - 1, i) \texttt{ ELSE } \bot,\\
e_a &:= \texttt{ite}(\texttt{op1}(0, \texttt{vr}), \texttt{op1}(2, \texttt{vr}),\\
&\ \texttt{ite}(\texttt{op1}(1, \texttt{vr}), \texttt{rec}(\texttt{op1}(3, \texttt{vr})),\\
&\ \texttt{rec}(\texttt{op2}(0, \texttt{vr}, \texttt{rec}(\texttt{op1}(4, \texttt{vr})))))).
\end{aligned}$$

It is proved in PVS that a computes the Ackermann function, i.e., for any $n, m, k \in \mathbb{N}$, $\mathit{ackermann}(m, n) = k$ if and only if $\varepsilon(a)(e_a, (n, m), (k, i))$, for some i, where *ackermann* is the recursive function defined in PVS as

$$\begin{aligned}
\mathit{ackermann}(m, n) := \ &\texttt{IF } m = 0 \texttt{ THEN } n + 1\\
&\texttt{ELSIF } n = 0 \texttt{ THEN } \mathit{ackermann}(m - 1, 1)\\
&\texttt{ELSE } \mathit{ackermann}(m - 1, \mathit{ackermann}(m, n - 1)).
\end{aligned}$$

In the definition of a, the type *Val* encodes the two inputs of the Ackermann function, but also the output of the function, which is given by the first entry of the second pair.

The proof of one of the implications in the statement of Example 1 proceeds by induction using a lexicographic order on (m, n). The other implication is proved using the induction schema generated for the inductive relation ε. Although it is not logically deep, this proof is tedious and long. However, it is mechanizable assuming that the PVS function and the PVS0 program share the same syntactical structure. As part of the work presented in this paper, a PVS strategy that automatically discharges equivalences between PVS functions and PVS0 programs was developed. This strategy is convenient since one of the objectives of this work is to reason about computational aspects of PVS functions through their embeddings in PVS0.

Example 1 also illustrates the use of built-in operators in PVS0. Despite its simplicity, this language is not minimal from a fundamental point of view. Indeed, since the type `T` is generic, any PVS function can be used as a building block in the construction of a PVS0 program. This feature is justified since all PVS functions are total. Therefore, they can be considered as atomic built-in operators. However, in contrast to proof assistants based on constructive logic, PVS allows for the definition of non-computable functions. The consequences of these features will be clear in the undecidability proof of the halting problem.

The following lemma states that the semantic evaluation relation ε is *deterministic*.

Lemma 1. *Let pvso be a program of type* PVSO[*Val*]. *For any expression e of type* PVSOExpr[*Val*] *and all values* $v_i, v'_o, v''_o \in \mathit{Val}$,

$$\varepsilon(\mathit{pvso})(e, v_i, v'_o) \text{ and } \varepsilon(\mathit{pvso})(e, v_i, v''_o) \text{ implies } v'_o = v''_o.$$

The proof of this lemma uses the induction schema generated for the inductive relation ε.

The relation ε is functional but not total, i.e., there are programs *pvso* and values v_i, for which there is no value v_o that satisfies $\varepsilon(\mathit{pvso})(\mathit{pvso}_e, v_i, v_o)$, where pvso_e is the program expression in *pvso*. This suggests the following definition of the *semantic termination predicate*.

$$T_\varepsilon(\mathit{pvso}, v_i) := \exists\, v_o \in \mathit{Val} \,:\, \varepsilon(\mathit{pvso})(\mathit{pvso}_e, v_i, v_o).$$

This predicate states that for a given program *pvso* and input v_i, the evaluation of the program's expression pvso_e on the value v_i terminates with the output value v_o. The program *pvso* is *total with respect to* ε if it satisfies the following predicate.[5]

$$T_\varepsilon(\mathit{pvso}) := \forall\, v \in \mathit{Val} \,:\, T_\varepsilon(\mathit{pvso}, v).$$

Semantic termination can also be specified by a function χ of type PVSO[*Val*] → (PVSOExpr[*Val*] $\times$ *Val* $\times\ \mathbb{N} \to \mathit{Val} \cup \{\Diamond\}$) defined as follows.

```
$\chi(O_1,O_2,\bot,e_f)(e,v_i,n)$ := IF $n=0$ THEN $\Diamond$ ELSE CASES $e$ OF
                 cnst($v$)  :  $v$;
                        vr  :  $v_i$;
                op1($j,e_1$)  :  IF $j<|O_1|$ THEN
                                 LET $v' = \chi(O_1,O_2,\bot,e_f)(e_1,v_i,n)$ IN
                                 IF $v' = \Diamond$ THEN $\Diamond$ ELSE $O_1(j)(v')$
                              ELSE $\Diamond$;
             op2($j,e_1,e_2$)  :  IF $j<|O_2|$THEN
                                 LET $v' \ = \chi(O_1,O_2,\bot,e_f)(e_1,v_i,n)$,
                                     $v'' = \chi(O_1,O_2,\bot,e_f)(e_2,v_i,n)$ IN
                                 IF $v' = \Diamond\ \vee v'' = \Diamond$ THEN $\Diamond$ ELSE $O_2(j)(v',v'')$
                              ELSE $\Diamond$;
                 rec($e_1$)  :  LET $v' = \chi(O_1,O_2,\bot,e_f)(e_1,v_i,n)$ IN
                              IF $v' = \Diamond$ THEN $\Diamond$ ELSE $\chi(O_1,O_2,\bot,e_f)(e_f,v',n-1)$;
           ite($e_1,e_2,e_3$)  :  LET $v' = \chi(O_1,O_2,\bot,e_f)(e_1,v_i,n)$ IN
                              IF $v' = \Diamond$ THEN $\Diamond$
                              ELSIF $v' \neq \bot$ THEN $\chi(O_1,O_2,\bot,e_f)(e_2,v_i,n)$
                              ELSE $\chi(O_1,O_2,\bot,e_f)(e_3,v_i,n)$.
```

In the definition of χ, n is the maximum number of nested recursive calls that are allowed in the evaluation of the recursive program for a given input. If this

[5] Polymorphism in PVS allow for the use of the same function or predicate name with different types.

limit is reached during an evaluation, the function χ returns the symbol $\diamondsuit$, which represents a "none" value. This function can be used to define an alternative predicate for semantic termination as follows.

$$T_\chi(pvso, v) := \exists\, n \in \mathbb{N} \,:\, \chi(pvso)(pvso_e, v, n) \neq \diamondsuit.$$

The program *pvso* is *total with respect to* χ if it satisfies the following predicate.

$$T_\chi(pvso) := \forall\, v \in Val \,:\, T_\chi(pvso, v).$$

The following theorem states that T_ε and T_χ captures the same notion of termination.

Theorem 1. *Let pvso be a PVS0 program of type* `PVS0`$[Val]$*. The following conditions hold:*

1. *For any $v_i \in Val$ and e of type* `PVS0Expr`$[Val]$*, $\varepsilon(pvso)(e, v_i, v_o)$ if and only if $v_o = \chi(pvso)(e, v_i, n)$, for some n, where $v_o \neq \diamondsuit$.*
2. *For any $v \in Val$, $T_\varepsilon(pvso, v)$ if and only if $T_\chi(pvso, v)$.*
3. *$T_\varepsilon(pvso)$ if and only if $T_\chi(pvso)$.*

In Theorem 1, Statement 2 and Statement 3 are consequences of Statement 1. Assuming $T_\chi(pvso, v)$, the proof of the Statement 1 requires the construction of the number $\mu(pvso, v) \in \mathbb{N}^+$ as follows.

$$\mu(pvso, v) := \min(\{n : \mathbb{N} \mid \chi(pvso)(pvso_e, v, n) \neq \diamondsuit\}).$$

This number satisfies the following property.

Lemma 2. *Let pvso be a program of type* `PVS0`$[Val]$ *and $v \in Val$ such that $T_\chi(pvso, v)$. For any $n \geq \mu(pvso, v)$, $\chi(pvso)(pvso_e, v, n) \neq \diamondsuit$.*

A PVS0 program *pvso* that satisfies $T_\chi(pvso)$ (or equivalently, $T_\varepsilon(pvso)$) is said to be *terminating*. The following lemma shows that there are non-terminating PVS0 programs.

Lemma 3. *Let $\Delta = (O_1, O_2, \bot, \texttt{rec}(\texttt{vr}))$. For any $v \in Val$, $\neg T_\chi(\Delta, v)$.*

The proof proceeds by showing that if $\chi(\Delta)(\Delta_e, v, n) \neq \diamondsuit$ for some v, where $n = \mu(\Delta, v)$, then it is also the case that $\chi(\Delta)(\Delta_e, v, n-1) \neq \diamondsuit$. By definition of μ, this is a contradiction.

Proving that a PVS0 program terminates using the semantic predicates $T_\varepsilon(pvso)$ and $T_\chi(pvso)$ is not very convenient. Indeed, these notions rely on actual computations of the program on a potentially infinite set of inputs. A syntactic criterion here called *Turing termination* relies on the existence of a well-founded relation $<$ over a type A and measure function $\mathcal{M}$ of type $Val \to A$ on the parameters of the recursive function such that $\mathcal{M}$ strictly decreases on every recursive call. This notion is adopted as the meta-theoretical definition of termination in several proof assistants. In PVS, this notion is implemented through the generation of the so called termination TCCs (Type Correctness

Conditions), where the measure function and the well-founded relation are provided by the user. This notion is formalized by defining, in PVS, an algorithm that generates termination TCCs for PVS0 programs. It has been proved that this Turing termination is equivalent to $T_\varepsilon(pvso)$ and, therefore, to $T_\chi(pvso)$. The formal infrastructure that is needed for defining Turing termination of PVS0 programs is out of the scope of the present work.

For a `PVS0` program $(O_1, O_2, \perp, e_f)$ and two values v_i and v_r, the definition of $v_i \to v_r$ is given by:

$$v_i \to v_r := \varepsilon(O_1, O_2, \perp, e_f)(e_f, v_i, v_r) \wedge \mathcal{M}(v_r) < \mathcal{M}(v_i)$$

The definition above relates an input value v_i of a PVS0 program and the argument v_r of a recursive call such that $\mathcal{M}(v_r) < \mathcal{M}(v_i)$. A key construction that is needed in the equivalence proof between the syntactic and the semantic termination criteria is the definition of the number $\Omega(v)$, where $v \in Val$, as follows.

$$\Omega(v) := \min(\{n : \mathbb{N}^+ \mid \forall\, v' \in V \,:\, \neg(v \to^n v')\}).$$

Intuitively, $\Omega(v)$ is the length of the longest path downwards starting from v. The following lemma states a relation between μ and Ω.

Lemma 4. *Let pvso be a PVS0 program that satisfies Turing termination for a well-founded relation $<$ over A and a measure function $\mathcal{M}$. For any value $v \in Val$, $\mu(pvso, v) \leq \Omega(v)$.*

3 Partial Recursive Functions

By design, the PVS0 language can directly encode any PVS function f of type $T \to T$, where T is an arbitrary PVS type. This feature enables the use of PVS functions as built-in operators in PVS0 program. The following lemma states that any PVS function can be embedded in a terminating PVS0 program.

Lemma 5. *Let f be a PVS function of type $T \to T$. The program $mk\_pvs0(f) = (O_1, O_2, \perp, e_f)$ of type* `PVS0`*, where $e_f = \texttt{op1}(0, \texttt{vr})$ and $O_1(0)(t) = f(t)$, satisfies the following properties:*

- *$mk\_pvs0(f)$ is terminating, i.e., $T_\varepsilon(mk\_pvs0(f))$.*
- *For any $t \in T$, $\varepsilon(mk\_pvs0(f))(e_f, t, f(t))$.*

The converse of Lemma 5 is not true in general as illustrated by the following theorem.

Lemma 6. *There is no PVS function f of type $T \to T$ such that for any $t \in T$, $\varepsilon(\Delta)(\texttt{rec}(\texttt{vr}), t, f(t))$, where Δ is the function defined in Lemma 3.*

However, any PVS0 program, even non-terminating ones, can be encoded as a PVS function of type $T \to T \cup \{\diamondsuit\}$, as stated in the following lemma.

Lemma 7. *Let pvso be a, possibly non-terminating, program of type* `PVS0` *and* pvso_e *the* `PVS0` *expression of pvso. The PVS function*

$$f(t) := \texttt{IF } T_\chi(\mathit{pvso}, t) \texttt{ THEN } \chi(\mathit{pvso})(\mathit{pvso}_e, t, \mu(\mathit{pvso}, t)) \texttt{ ELSE } \diamondsuit$$

satisfies the following property for any $t \in T$.

$$T_\varepsilon(\mathit{pvso}, t) \textit{ if and only if } \varepsilon(\mathit{pvso})(\mathit{pvso}_e, t, f(t)).$$

The proofs of the previous lemmas are straightforward applications of the definitions of T_ε, ε, T_χ, and χ.

A consequence of Lemma 5 and Lemma 7 is that is possible to define an oracle of type `PVS0`[*Val*], where *Val* = `PVS0`, that decides if a program of type `PVS0` is terminating or not. The existence of that oracle is stated in the following theorem.

Theorem 2. *The program Oracle* $= (O_1, O_2, \bot, e_f)$ *of type* `PVS0`[*Val*], *where* *Val* = `PVS0`, $e_f = \mathit{mk\_pvs0}(\texttt{LAMBDA}(\mathit{pvso} : \texttt{PVS0}) : \texttt{IF } T_\varepsilon(\mathit{pvso}) \texttt{ THEN } \top \texttt{ ELSE } \bot)$, *and* $\top \neq \bot$, *has the following properties.*

- *Oracle is a terminating* `PVS0`[*Val*] *program, i.e.,* $T_\varepsilon(\mathit{Oracle})$.
- *Oracle decides termination of any* `PVS0` *program, i.e., for any pvso of type* `PVS0`,

$$\chi(\mathit{Oracle})(\mathit{Oracle}_e, \mathit{pvso}, \mu(\mathit{Oracle}, \mathit{pvso})) = \top \textit{ if and only if } T_\varepsilon(\mathit{pvso}).$$

This counterintuitive result is possible because PVS, in contrast to proof assistants based on constructive logic, allows for the definition of total functions that are non-computable, e.g., $\texttt{LAMBDA}(\mathit{pvso} : \texttt{PVS0}) : \texttt{IF } T_\varepsilon(\mathit{pvso}) \texttt{ THEN } \top \texttt{ ELSE } \bot$. These non-computable functions can be used in the construction of terminating PVS0 programs through the built-in operators. Therefore, in order to formalize the notion of partial recursive functions in PVS0, it is necessary to restrict the way in which programs are built.

First, the basic type T is set to $\mathbb{N}$, i.e., $\mathit{Val} = \mathbb{N}$, where the number 0 represents the value false, i.e., $\bot = 0$. Any value different from 0 represents a true value, in particular $\top = 1$. Second, the built-in operators used in the construction of programs are restricted by a hierarchy of levels: the operators in the first level can only be defined using projections ($\Pi_1(x, y : \mathbb{N}) := x$ and $\Pi_2(x, y : \mathbb{N}) := y$), successor ($\mathit{succ}(x : \mathbb{N}) : x + 1$), and greater or equal than ($\mathit{ge}(x, y : \mathbb{N}) : \texttt{IF } x \geq y \texttt{ THEN } \top \texttt{ ELSE } \bot$) functions. Operators in higher levels can only be constructed using programs from the previous level. This idea is formalized by the predicate $\mathit{pvs0\_level} : \mathbb{N} \to \mathtt{PVS0}[\mathbb{N}] \to \mathtt{bool}$ as shown below.

```
$\mathit{pvs0\_level}(n)(O_1, O_2, \perp, e_f) :=$
  IF $n = 0$ THEN $O_1 = \langle \mathit{succ} \rangle \ \wedge O_2 = \langle \mathit{\Pi}_1, \mathit{\Pi}_2, ge \rangle$
  ELSE ( $\exists p' \in \mathtt{PVS0}[\mathbb{N}]\ :\ \mathit{pvs0\_level}(n-1)(p') \wedge$
              LET $({O_1}', {O_2}', \perp', {e_f}') = p',\ l'_1 = |{O_1}'|$ IN
                $|O_1| = l'_1 + 1 \ \wedge$
                $(\ \forall i \in \mathbb{N} : i < l'_1 \Rightarrow O_1(i) = {O_1}'(i)\ ) \wedge$
                $(\ \forall v \in \mathbb{N} : \varepsilon(p')({e_f}', v, O_1(l'_1)(v))\ )\ ) \wedge$
         ( $\exists p' \in \mathtt{PVS0}[\mathbb{N}] :\ \mathit{pvs0\_level}(n-1)(p') \wedge$
             LET $({O_1}', {O_2}', \perp', {e_f}') = p', l'_2 = |{O_2}'|$ IN
              $|O_2| = l'_2 + 1 \ \wedge$
              $(\ \forall i \in \mathbb{N} : i < l'_2 \Rightarrow O_2(i) = {O_2}'(i)\ ) \wedge$
              $(\ \forall v_1, v_2 \in \mathbb{N} : \varepsilon(p')({e_f}', \kappa_2(v_1, v_2), O_2(l'_2)(v_1, v_2))\ )\ ),$
```

where the function κ_2 is an encoding of pairs of natural numbers onto natural numbers defined as $\kappa_2(m, n) := (m + n + 1) \times (m + n)/2 + n$.

The type `PartialRecursive` is defined to be a subtype of `PVS0`$[\mathbb{N}]$ containing all the programs *pvso* such that there is a natural n for which $\mathit{pvs0\_level}(n)(\mathit{pvso})$ holds. Additionally, `Computable` is a subtype of `PartialRecursive` containing those elements that are also terminating according to the aforementioned definitions.

The following theorem states that `PartialRecursive`, and thus `Computable`, are enumerable types.

Theorem 3. *There exists a PVS function of type* $\mathbb{N} \to$ `PartialRecursive` *that is surjective.*

As a corollary of this theorem, the inverse of this surjective function, denoted as κ_P is an injective function of type `PartialRecursive` $\to \mathbb{N}$.

The proof of Theorem 3 is technically involved. The proof proceeds by showing that each level is enumerable. Therefore, the function κ_P exists since the countable union of countable sets is also countable. A similar work is presented in [4] by Foster an Smolka. They encode a lambda term into another lambda term such that it represents a natural number using the Scott numbers codification and use that encoding to formalize in Coq the Rice's Theorem.

The function κ_P is used in the proof of the undecidability of the halting problem for PVS0 to encode `PartialRecursive` programs as inputs of a `Computable` program. This function is a key element in the construction of the self-reference argument used in the diagonalization approach of the undecidability proof.

4 Undecidability of the Halting Problem

The classical formalization of the undecidability of the halting problem starts by assuming the existence of an oracle capable of deciding whether a program halts for any input. A Gödelization function transforms the tuple of program and input into a single input to the oracle. After that, using the oracle, another program is created such that if the encoded program halts it enters into an infinite loop. Otherwise, it produces an answer and halts. Passing this program as an input to itself results in the expected contradiction.

Here, the undecidability of the halting problem was formalized using the notions of termination for PVS0 defined in the previous section and the Cantor's *diagonalization* technique.

Theorem 4 (Undecidability of the Halting Problem for PVS0). *There is no program* $oracle = (O_1, O_2, \bot, e_o)$ *of type* `Computable` *such that for all* $pvso = ({O_1}', {O_2}', \bot, e_f)$ *of type* `PartialRecursive` *and for all* $n \in \mathbb{N}$*,*

$$T_\varepsilon(pvso, n) \textit{ if and only if } \neg\varepsilon(oracle)(e_o, \kappa_2(\kappa_P(pvso), n), \bot).$$

Proof. The proof proceeds by assuming the existence of an oracle to derive a contradiction. Suppose there exists a program $oracle = (O_1, O_2, \bot, e_o)$ of type `Computable` such as the one presented in the statement of the theorem. Then, a $\texttt{PVS0}[\mathbb{N}]$ program $pvso = ({O_1}', {O_2}', \bot, e_f)$ can be defined, where $O_1'(k) = O_1(k)$, for $k < |O_1|$, $O_2'(k) = O_2(k)$, for $k < |O_2|$, and

- ${O_1}'(|O_1|)(i) = choose(\{a : \mathbb{N} \mid \varepsilon(oracle)(e_o, i, a)\})$,
- ${O_2}'(|O_2|)(i, j) = choose(\{a : \mathbb{N} \mid \varepsilon(oracle)(e_o, \kappa_2(i, j), a)\})$, and
- $e_f = \texttt{ite}(\texttt{op2}(|O_2|, \texttt{vr}, \texttt{vr}), rec(\texttt{vr}), \texttt{vr})$,

The PVS function *choose* returns an arbitrary element from a non-empty set. The sets used in the definitions of O_1' and O_2' are non-empty since *oracle* is `Computable` and, therefore, terminating. The program *pvso* is built in such a way that it belongs to the next level from the level of *oracle*.

Let n be the natural number $\kappa_P(pvso)$. The rest of the proof proceeds by case analysis.

- **Case1** : $\varepsilon(oracle)(e_o, \kappa_2(n, n), \bot)$**.** This case holds if and only if $\neg T_\varepsilon(pvso, n)$. Expanding T_ε one obtains

$$\begin{aligned} \neg\exists(v : \mathbb{N}) : \exists(v_o : \mathbb{N}) : \ & \varepsilon(pvso)(\texttt{op2}(|O_2|, \texttt{vr}, \texttt{vr}), n, v_o) \wedge \\ & \texttt{IF } v_o \neq \bot \\ & \texttt{THEN } \varepsilon(pvso)(rec(\texttt{vr}), n, v) \\ & \texttt{ELSE } \varepsilon(pvso)(\texttt{vr}, n, v). \end{aligned} \tag{1}$$

  Expanding ε in $\varepsilon(pvso)(\texttt{op2}(|O_2|, \texttt{vr}, \texttt{vr}), n, v_o)$ yields

$$choose(\{a : \mathbb{N} \mid \varepsilon(oracle)(e_o, \kappa_2(n, n), a)\}) = v_o.$$

  Since $\varepsilon(oracle)(e_o, \kappa_2(n, n), \bot)$ holds, $\bot = v_o$. Therefore, Formula (1) is equivalent to

$$\neg\exists(v : \mathbb{N}) : \varepsilon(pvso)(\texttt{vr}, n, v). \tag{2}$$

  The predicate $\varepsilon(pvso)(\texttt{vr}, n, v)$ holds if and only if $n = v$. Hence, Formula (2) states that $\neg\exists(v : \mathbb{N}) : n = v$, where n is a natural number. This is a contradiction.

- **Case2** : $\neg\varepsilon(\mathit{oracle})(e_o, \kappa_2(n,n), \bot)$. This case holds if and only if $T_\varepsilon(\mathit{pvso}, n)$. From Theorem 1, $T_\chi(\mathit{pvso}, n)$. If the proof starts directly from $T_\varepsilon(\mathit{pvso}, n)$, after expanding and simplifying it, $T_\varepsilon(\mathit{pvso}, n)$ is obtained once again, which implies that there is not such an n, giving a contradiction. However, since PVS does not accept the definition of a function that enters into such an infinite loop, the solution is to apply the equivalence Theorem 1. Expanding the definition of T_χ yields

  $$\exists\, m \in \mathbb{N} \,:\, \chi(\mathit{pvso})(\mathtt{ite}(\mathtt{op2}(|O_2|, \mathtt{vr}, \mathtt{vr}), \mathit{rec}(\mathtt{vr}), \mathtt{vr}), n, m) \neq \diamondsuit.$$

  If there exists such m, it can be chosen as the minimal natural that makes the above proposition hold. Expanding the definition of χ yields

  $$\begin{pmatrix} \texttt{IF}\ \chi(\mathit{pvso})(\mathtt{op2}(|O_2|, \mathtt{vr}, \mathtt{vr}), n, m) \neq \diamondsuit\ \texttt{THEN} \\ \quad \texttt{IF}\ \chi(\mathit{pvso})(\mathtt{op2}(|O_2|, \mathtt{vr}, \mathtt{vr}), n, m) \neq \bot\ \texttt{THEN} \\ \quad\ \chi(\mathit{pvso})(\mathit{rec}(\mathtt{vr}), n, m) \\ \quad\ \texttt{ELSE}\ \chi(\mathit{pvso})(\mathtt{vr}, n, m) \\ \texttt{ELSE}\ \diamondsuit \end{pmatrix} \neq \diamondsuit. \qquad (3)$$

  If the condition of the first if-then-else were false, then Formula (3) reduces to $\diamondsuit \neq \diamondsuit$, which is a contradiction. Therefore, this condition must be true. After expanding and simplifying χ, $\chi(\mathit{pvso})(\mathtt{op2}(|O_2|, \mathtt{vr}, \mathtt{vr}), n, m)$ reduces to

  $$\mathit{choose}(\{a : \mathbb{N} \mid \varepsilon(\mathit{oracle})(e_o, \kappa_2(n,n), a)\}).$$

  Let $v = \mathit{choose}(\{a : \mathbb{N} \mid \varepsilon(\mathit{oracle})(e_o, \kappa_2(n,n), a)\})$. If $v = \bot$, then

  $$\varepsilon(\mathit{oracle})(e_o, \kappa_2(n,n), \bot).$$

  This is a contradiction since $n = \kappa_P(\mathit{pvso})$.
  Thus, $\chi(\mathit{pvso})(\mathtt{op2}(|O_2|, \mathtt{vr}, \mathtt{vr}), n, m) \neq \bot$. Then, Formula (3) can be simplified to

  $$\chi(\mathit{pvso})(\mathit{rec}(\mathtt{vr}), n, m) \neq \diamondsuit.$$

  Finally, expanding χ results in $\chi(\mathit{pvso})(e_f, n, m-1) \neq \diamondsuit$. This contradicts the minimality of m, completing the proof.

□

5 Related Work

Formalization of models of computation is not a novelty. In recent work, Foster and Smolka [4] formalized, in Coq, Rice's Theorem, which states that non-trivial semantic properties of procedures are undecidable. This theorem is a variant of Post's Theorem, which states that a class of procedures is decidable if it is recognizable, co-recognizable, and logically decidable, and the fact that the class of total procedures is not recognizable. This work was done for a model of a functional language presented as a weak call-by-value lambda-calculus in which

β-reduction can be applied only if the redex is not below an abstraction and if the argument is an abstraction. Results are adaptable for call-by-value functional languages and Turing-complete models formalized in Coq. Larchey-Wendling [6] gave a formalization that "the Coq type $\texttt{nat}^k \to \texttt{nat}$ contains every recursive function of arity k which can be proved total in Coq." There, `nat` is the Coq type for Peano naturals. This is a class of functions between the primitive functions $\mathbb{N}^k \to \mathbb{N}$ and the partial recursive functions $\mathbb{N}^k \rightharpoondown \mathbb{N}$. Proving that the former class of functions is a set of terms of Coq type $\mathbb{N}^k \to \mathbb{N}$ is simple since Coq includes all required recursive schemes (basic functions plus composition and recursion). The paper advances into characterizing the type $\mathbb{N}^k \rightharpoondown \mathbb{N}$, which is not straightforward and is related to the class of terminating functions.

Highly relevant related papers include [9] in which Norrish presented a formalization in HOL4 of several results of computability theory using as models recursive functions and the λ-calculus. The mechanizations include proofs of the equivalence of the computational power of both models, undecidability of the halting problem, existence of universal machines, and Rice's Theorem in the λ-calculus. In addition, in [14] Xu, Zhang and Urban presented a formalization of computability theorems in Isabelle/HOL using as models Turing machines, abacus machines (a kind of register machines) and recursive functions. Formalized results include also undecidability of the halting problem and existence of universal Turing machines.

In contrast to those approaches, this paper deals with a computational model that is specified as a concrete functional programing language, namely PVS0. For this language, all the elements required in a classic-style proof of undecidability of termination, such as *Gödelization* of programs and inputs, are developed. Having a concrete programming language such as PVS0 enables the formalization and comparison of different termination analysis techniques for this language. In fact, the current formalization is part of a larger library that relates different termination criteria such as semantic termination, Turing termination [13], dependency pairs [1, 15, 2], and techniques based on the size change principle [7, 11, 5] such as calling context graphs [8] and matrix-weighted graphs [3].

6 Conclusion and Future Work

This paper presents the formalization of the undecidability of the halting problem for a simple functional language called PVS0. This formalization required the definition of several notions of termination, which are all proven to be equivalent. Since PVS0 generally allows for the encoding of non-computable functions through the use built-in operators, the undecidability proof is done on a restriction of PVS0 programs. First, the input type is restricted to natural numbers. Then, `PartialRecursive` and `Computable` programs are constructed using a layered approach where programs at level $n+1$ can only depend on programs at level n or below. The existence of a surjective mapping from natural numbers to PVS0 programs constructed using this layered approach is formally verified. This mapping enables the definition of a Gödelization function for the type of programs PartialRecursive, which is crucial in the undecidability proof of the halting problem.

PVS0 is used to study termination and totality properties of PVS functions with the objective of automating the generation of measure functions in PVS recursive definitions. As part of this study, termination analysis techniques for PVS0 programs have been formalized and verified. Furthermore, strategies that automatically prove the equivalence between PVS recursive functions and PVS0 programs and that discharge termination TCCs of PVS recursive functions through their PVS0 counterpart have been developed. Future work includes extending the syntax of the PVS0 minimal language to support constructs such as let-in expressions and extending the proposed framework to support higher-order functions.

Finally, it is conjectured that the PVS0 language is Turing-complete. The proof of this property, which is work in progress, is not conceptually difficult but is technically involved. For simplicity, the PVS0 language only allows for the definition of one recursive function. Composition could be encoded using the lists of built-in operators. However, by definition, built-in operators are terminating. Therefore, to enable composition of (possibly) non-terminating programs, it is necessary to encode it in PVS0 itself using, among other things, the Gödelization of programs. This encoding increases the complexity of the proof of this conjecture.

References

1. Thomas Arts. Termination by absence of infinite chains of dependency pairs. In *Proceedings Trees in Algebra and Programming - CAAP 1996, 21st International Colloquium*, volume 1059 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 1996.
2. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.
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. Yannick Forster and Gert Smolka. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *Proceedings Interactive Theorem Proving - 8th International Conference, ITP 2017*, volume 10499 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017.
5. Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle Functions via Termination of Rewriting. In *Proceedings Interactive Theorem Proving - Second International Conference, ITP 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2011.
6. Dominique Larchey-Wendling. Typing Total Recursive Functions in Coq. In *Proceedings Interactive Theorem Proving - 8th International Conference, ITP 2017*, volume 10499 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2017.
7. 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.

8. Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *Proceedings Computer Aided Verification CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414, 2006.
9. Michael Norrish. Mechanised computability theory. In *Proceedings Interactive Theorem Proving - Second International Conference, ITP 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.
10. 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.
11. René Thiemann and Jürgen Giesl. Size-change termination for term rewriting. In Robert Nieuwenhuis, editor, *Proceedings Rewriting Techniques and Applications, 14th International Conference, RTA 2003*, volume 2706 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2003.
12. Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42(1):230–265, 1937.
13. Alan M. Turing. Checking a large routine. In Martin Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.
14. Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising turing machines and computability theory in isabelle/hol. In *Proceedings Interactive Theorem Proving - 4th International Conference, ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.
15. Akihisa Yamada, Christian Sternagel, René Thiemann, and Keiichirou Kusakari. AC dependency pairs revisited. In *Proceedings 25th EACSL Annual Conference on Computer Science Logic, CSL 2016*, volume 62 of *LIPIcs*, pages 8:1–8:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.