Formalization of Rice's Theorem over a Functional Language Model

Thiago Mendonça Ferreira Ramos* and Ariane Alves Almeida* and Mauricio Ayala-Rincón*,†

*Departments of † Mathematics and * Computer Science*
Universidade de Brasília
Brasília D.F., Brazil

Abstract

Classical proofs of Rice's Theorem assume the existence of a universal (Turing) machine and build a reduction from the problem of deciding whether a machine halts or not to the problem of separability of semantic properties of machines. This work presents a formalization in PVS of Rice's Theorem over a computational model given as a class of partial recursive functions. The main differences with classical proofs are that the given formalization is developed over a functional programming model and that the proof does not depends on the undecidability of the Halting Problem being made directly without using any translation to or from other computational models. As corollaries, straightforward formalizations of the undecidability of the Halting Problem, functional equivalence problem, existence of fixed points problem and self-replication problem are obtained.

Keywords: Functional Programming Models, Automating Termination, Computability Theory, Halting Problem, Rice's Theorem, Theorem Proving, PVS

1 Introduction

In computer science, it is well-known that all Turing complete models of computation have some expressiveness limits, this means that not all kind of *problems* can be solved and that *programs* over each of these models with the same or different semantics cannot be distinguished. One of these limits is given by Rice's Theorem. This result says that it is impossible to build a program that decides a semantic predicate over other programs, unless the predicate is either the set of all programs or the empty set.

The formalization is performed in the proof assistant PVS, using as model of computation a variant of a first-order functional language called PVS0. In this language there are constants, only one variable, unary and binary built-in operators, if-then-else instructions and recursive calls. PVS0 was designed with the main aim of formalizing the correction of mechanisms to automate verification of termination in PVS, which makes of high interest the exploration of the computability theory of PVS0 as a model of computation. Indeed, PVS0 is applied to formalize the equivalence among different criteria of termination, including size change principle [13] termination criteria such as Manolios and Vroom's Calling Context Graphs [14] and Avelar et al Matrix Weighted Graphs [3], as well as Turing termination [17] (that is the criterion applied for the PVS specification language), and Dependency Pairs termination [1][2]. The libraries for PVS0, which include equivalence proofs of these termination criteria, are available as part of the NASA LaRC PVS library.

The aim of this paper is related with a second relevant objective that is the study of PVS0 as a model of computation, for which it is required the formalization of its *computational theory.* In addition to verification of termination criteria over PVS0, it is important to formalize computability properties of the model to certify that it is indeed a reasonable and expressive model of computation. Previous work presented the formalization of the undecidability of the Halting Problem for the PVS0 model [5]. In that work this result was formalized for a language model different from the one used in the current paper, which allows only programs that consist of a unique (recursive) function, called here the single-function PVS0 model. The language model used in this paper is more realistic, accepting, similarly to functional specifications, a list of functions such that each function can call and be called by all functions in the list using their indices. This model is called the multiple-function PVS0 model (or just PVS0 when no confusion arise). The undecidability of the Halting Problem was formalized for the multiple-function model too, but this result is now obtained just as a corollary of the formalization of Rice's theorem.

The PVS0 language model allows a unique input/output type, passed as a parameter of the PVS theory, and for dealing with Rice's Theorem the type of naturals is used. Basic operators including successor, greater than and projections, provide a Turing Complete model. The formalization assumes that there is a manner of computing the own number of Gödel of each PVS0 program, i.e., the recursion theorem. Furthermore, it follows Cantor's diagonal argument, similarly to standard proofs of the undecidability of the Halting Problem and uncountability of real numbers. An alternative formalization approach could be based on the construction of a universal program for the PVS0 model, but it would increase the complexity of the formalization. Using such construction, the proof would require reducing the Halting Problem to the problem of separability of semantic properties of PVS0 programs, which is not the case of the current formalization. Thus, the current formalization depends only on the above mentioned assumption and does not uses undecidability of the Halting Problem. In textbooks, assumptions such as the existence of universal machines and programs as well as Gödelization of machines and programs are intuitively given without providing complete constructions. In order to rule out the assumption and get a full formalization, the above mentioned basic operators are required to guarantee Turing completeness and thus the recursion theorem.

After giving the semantics of the PVS0 model and the specifications of computable and partial recursive classes in Sections 2 and 3, resp., Section 4 explains the formalization of Rice's Theorem. Section 5 discusses crucial results formalized as simple corollaries of Rice's Theorem including the undecidability of the Halting Problem. Then, before concluding and discussing current and future work in Section 7, Section 6 discusses related work.

The formalization of Rice's Theorem is available at

`https://github.com/thiagomendoncaferreiraramos/ricestheorem`

It requires the installation of the NASA library `https://github.com/nasa/pvslib`, which includes the PVS0 library for termination criteria and their equivalence over the PVS0 single-function model.

2 Semantics of PVS0 Programs

Expressions of the PVS0 functional language have the grammar below.

$$\begin{aligned} expr ::= \; & \texttt{cnst}(T) \mid \\ & \texttt{vr} \mid \\ & \texttt{op1}(\mathbb{N}, expr) \mid \\ & \texttt{op2}(\mathbb{N}, expr, expr) \mid \\ & \texttt{rec}(\mathbb{N}, expr) \mid \\ & \texttt{ite}(expr, expr, expr) \end{aligned}$$

Above, T is an uninterpreted type over which PVS0 expressions are interpreted. Constants of type T have grammar $\texttt{cnst}(T)$ and the symbol `vr` is the unique symbol of variable. `op1` and `op2` denote respectively unary and binary operators indexed by naturals. The symbol `rec` is for recursive calls and is also indexed by naturals used to select the function in a PVS0 program to be called, applying it to the result of the evaluation of the second argument, *expr*. Finally, `ite` is the symbol of the branching instruction and its evaluation has the same semantics as the instruction if-the-else.

The kernel of a PVS0 program is a non empty list of PVS0 expressions whose main expression (function) is the one in the head of the list and, for which the symbol of recursive calls, $\texttt{rec}(i, \_)$, interprets functions calls as evaluations of the i^{th} PVS0 expression in the list. PVS0 programs include also lists of unary and binary functions to interpret the symbols of unary and binary operators, and an element of the worked type to be interpreted as false (for the evaluation of the guards of `ite` instructions). Thus, PVS0 programs are 4-tuples of the form $(O_1, O_2, \bot, E_f)$, where O_1 and O_2 are the lists of unary and binary functions, $\bot$ is an element of T to be interpreted as false, and E_f is the kernel of the program.

The (eager) evaluation predicate ε, for PVS0 programs is defined below.

$$
\begin{array}{rcl}
\varepsilon(O_1, O_2, \bot, E_f)(e, v_i, v_o) & := & \texttt{CASES}\ e\ \texttt{OF} \\
\texttt{cnst}(v) & : & v_o = v; \\
\texttt{vr} & : & v_o = v_i; \\
\texttt{op1}(j, e_1) & : & \exists\,(v' : T) : \\
 & & \varepsilon(O_1, O_2, \bot, E_f)(e_1, v_i, v') \wedge \\
 & & \texttt{IF}\ \ j < |O_1|\ \texttt{THEN} \\
 & & \quad v_o = O_1(j)(v'); \\
 & & \texttt{ELSE}\ \ v_o = \bot \\
\texttt{op2}(j, e_1, e_2) & : & \exists\,(v', v'' : T) : \\
 & & \varepsilon(O_1, O_2, \bot, E_f)(e_1, v_i, v') \wedge \\
 & & \varepsilon(O_1, O_2, \bot, E_f)(e_2, v_i, v'') \wedge \\
 & & \texttt{IF}\ \ j < |O_2|\ \texttt{THEN} \\
 & & \quad v_o = O_2(j)(v', v''); \\
 & & \texttt{ELSE}\ \ v_o = \bot \\
\texttt{rec}(j, e_1) & : & \exists\,(v' : T) : \varepsilon(O_1, O_2, \bot, E_f)(e_1, v_i, v') \wedge \\
 & & \texttt{IF}\ \ j < |E_f|\ \texttt{THEN} \\
 & & \quad \varepsilon(O_1, O_2, \bot, E_f)(E_f(j), v', v_o) \\
 & & \texttt{ELSE}\ \ v_o = \bot \\
\texttt{ite}(e_1, e_2, e_3) & : & \exists\,(v' : T) : \varepsilon(O_1, O_2, \bot, E_f)(e_1, v_i, v') \wedge \\
 & & \texttt{IF}\ v' \neq \bot\ \texttt{THEN}\ \varepsilon(O_1, O_2, \bot, E_f)(e_2, v_i, v_o) \\
 & & \texttt{ELSE}\ \varepsilon(O_1, O_2, \bot, E_f)(e_3, v_i, v_o).
\end{array}
$$

Parameters v_i and v_0 of ε are the input and output values. Note that the parameter e is needed since the evaluation of a program $(O_1, O_2, \bot, E_f)$, for short denoted as *pvso*, leads to evaluation of sub expressions e of the expressions in E_f and, when a recursive call, as $\texttt{rec}(j, e_1)$ in the recursive case above, is evaluated, the expression $E_f(j)$ should be considered. Using the ε predicate, another predicate is defined that holds for PVS0 programs and correct inputs and outputs, specified as below, where $\mathit{pvso}'4$ denotes the fourth element of *pvso*.

$$\gamma(\mathit{pvso})(v_i, v_o) = \varepsilon(\mathit{pvso})(\mathit{pvso}'4(0), v_i, v_o)$$

Note that γ starts the evaluation from the main function of the program that is $\mathit{pvso}'4(0)$, the head of the list of expressions $\mathit{pvso}'4$, the same as E_f.

In order to show results related with automation of termination, semantic termination of PVS0 programs was specified in [5]. This definition is required to prove completeness and equivalence of practical termination criteria as well as to formalize computablity results such as undecidability of the Halting Problem and Rice's theorem. Termination is specified as the *semantic termination predicate* below.

$$T_\varepsilon(\mathit{pvso}, v_i) := \exists\,(v_o : T)\ :\ \varepsilon(\mathit{pvso})(\mathit{pvso}'4(0), v_i, v_o).$$

This predicate states that for a given program *pvso* and input v_i, the evaluation of the program's expression $\mathit{pvso}'4(0)$ 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 (here notice that polymorphism in PVS allows the use of the same function or predicate name with different types).

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

Both the formalizations of the undecidability of the Halting Problem as given in [5] and Rice's Theorem in this paper require to build composition of functions and programs to give rise to contradictions. Using the multiple-function PVS0 model in this work, it is possible to specify the composition of arbitrary PVS0 functions, but for the single-function model used in [5] it was not the case. The required compositions in [5] were specified manually as new single-function PVS0 programs, which was possible since the specific functions to be composed were always terminating.

To compose PVS0 programs that share the same unary and binary operators and interpretation of false, a fundamental issue is the notion of *offset* used to adjust the indices of function calls in program lists. It works as a simplified version of offset in assembly languages, where instruction labels in some piece of code are adjusted when they are shifted. For PVS0 programs this is specified as the function β.

$$\begin{aligned}
\beta(n)(e) &:= \texttt{CASES}\ e\ \texttt{OF}\\
\texttt{cnst}(v) &: \texttt{cnst}(v);\\
\texttt{vr} &: \texttt{vr};\\
\texttt{op1}(j,e_1) &: \texttt{op1}(j,\beta(n)(e_1));\\
\texttt{op2}(j,e_1,e_2) &: \texttt{op2}(j,\beta(n)(e_1),\beta(n)(e_2));\\
\texttt{rec}(j,e_1) &: \texttt{rec}(j+n,\beta(n)(e_1));\\
\texttt{ite}(e_1,e_2,e_3) &: \texttt{ite}(\beta(n)(e_1),\beta(n)(e_2),\beta(n)(e_3))
\end{aligned}$$

Using β, the composition of $(O_1, O_2, \bot, A)$ and $(O_1, O_2, \bot, B)$, two PVS0 programs, is expressed by the following property.

$$\forall(v_i, v_o) : \exists(v) : \gamma(O_1, O_2, \bot, B)(v_i, v) \wedge \gamma(O_1, O_2, \bot, A)(v, v_o) \Leftrightarrow$$

$$\gamma(O_1, O_2, \bot, [\texttt{rec}(1, \texttt{rec}(1 + |A|, \texttt{vr}))] :: \mathit{map}(\beta(1))(A) :: \mathit{map}(\beta(1 + |A|))(B))(v_i, v_o)$$

As an example, consider the unitary lists of unary and binary operators below for predecessor and multiplication on $\mathbb{N}$, and element to interpret as false.

- $O_1 := [\lambda(n : \mathbb{N}^+) : n - 1]$
- $O_2 := [\lambda(n, m : \mathbb{N}) : n \times m]$
- $\bot := 0$

Consider now the following unitary lists of expressions, where the first one specifies the basic function quadratic and the second one the factorial function.

- $\mathit{quadratic} := [\texttt{op2}(0, \texttt{vr}, \texttt{vr})]$
- $\mathit{factorial} := [\texttt{ite}(\texttt{vr}, \texttt{op2}(0, \texttt{vr}, \texttt{rec}(0, \texttt{op1}(0, \texttt{vr}))), \texttt{cnst}(1))]$

The 4-tuples $(O_1, O_2, \bot, \mathit{quadratic})$ and $(O_1, O_2, \bot, \mathit{factorial})$ are then PVS0 programs for these functions. The correction of the construction of the program for the composition of *factorial* and *quadratic* specified using β, is expressed as the property below.

$$\begin{aligned}
\forall(v_i) : \gamma(O_1, O_2, \bot,\ & [\texttt{rec}(1, \texttt{rec}(1 + |\mathit{factorial}|, \texttt{vr}))] ::\\
& \mathit{map}(\beta(1))(\mathit{factorial}) ::\\
& \mathit{map}(\beta(1 + |\mathit{factorial}|))(\mathit{quadratic}))(v_i, (v_i^2)!)
\end{aligned}$$

Since the result of $|\mathit{factorial}|$ is 1, $\mathit{map}(\beta(1))(\mathit{factorial})$ is $\texttt{ite}(\texttt{vr}, \texttt{op2}(0, \texttt{vr}, \texttt{rec}(1, \texttt{op1}(0, \texttt{vr}))), \texttt{cnst}(1))$ and $\mathit{map}(\beta(1 + |\mathit{factorial}|))(\mathit{quadratic})$ is $\texttt{op2}(0, \texttt{vr}, \texttt{vr})$, the simplification of the 4-tuple written above is :

$$(O_1, O_2, \bot, [\texttt{rec}(1, \texttt{rec}(2, \texttt{vr}))] :: [\texttt{ite}(\texttt{vr}, \texttt{op2}(0, \texttt{vr}, \texttt{rec}(1, \texttt{op1}(0, \texttt{vr}))), \texttt{cnst}(1))] :: [\texttt{op2}(0, \texttt{vr}, \texttt{vr})])$$

A functional alternative for the semantic evaluation predicate ε should take into consideration the case in which the evaluation does not return an output. The problem is solved by adding an element to the working type that is interpreted as *none* and denoted by $\diamondsuit$. In addition, the evaluation function includes a parameter that limits the allowed number of nested recursive calls: when this limit is reached, the function returns $\diamondsuit$. This is specified as the function χ below.

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

The predicate ε and function χ are proved equivalent in the following sense:

$$\forall(pvso, e, v_i, v_o) : \varepsilon(pvso)(e, v_i, v_o) \Leftrightarrow \exists(n) : \chi(pvso)(n, pvso'4(0), v_i) = v_o \wedge v_o \neq \diamondsuit$$

Similarly to [5], a terminating program, *pvso*, satisfies the predicate:

$$\forall(v_i) : \exists(v_o) : \gamma(pvso)(v_i, v_o) \text{ or equivalently } \forall(v_i) : \exists(n) : \chi(pvso)(n, pvso'4(0), v_i) \neq \diamondsuit$$

To define the classes of partial recursive and computable functions, indices of the function calls in PVS0 programs are restricted to valid indices:

$$valid\_index\_rec(e, n) := \forall(i, e_1) : subterm(\texttt{rec}(i, e_1), e) \Rightarrow i < n$$

$$valid\_index(E_f) := \forall(i < |E_f|) : valid\_index\_rec(E_f(i), |E_f|)$$

3 PVS0 Computable and Partial Recursive Classes

Let $\boldsymbol{O_1}$, $\boldsymbol{O_2}$ and $\bot$ be fixed unary, binary operators and false symbol and, besides that, fix the input and output type as naturals. Below, it is defined a class of partial recursive functions.

$$partial\_recursive(pvso) := pvso'1 = \boldsymbol{O_1} \ \wedge \ pvso'2 = \boldsymbol{O_2} \ \wedge \ pvso'3 = \bot \ \wedge \\ valid\_index(pvso'4)$$

If *partial_recursive*(*pvso*) holds, *pvso* is of type `PartialRecursive`. If in addition *pvso* is terminating it is of type `Computable`.

To prove Rice's Theorem, it is necessary to formalize some lemmas about code shifting. The first lemma is:

$$\forall(O_1, O_2, \bot, A, B, e, v_i, n) : \\ \chi(O_1, O_2, \bot, B)(e, v_i, n) = \chi(O_1, O_2, \bot, A :: map(\beta(|A|))(B))(\beta(|A|)(e), v_i, n)$$

The second lemma is:

$$\forall(O_1, O_2, \perp, B, v_i, n) : \forall(e \mid \mathit{valid\_index\_rec}(e, |A|)) : \forall(A \mid \mathit{valid\_index}(A)) :$$
$$\chi(O_1, O_2, \perp, A)(e, v_i, n) = \chi(O_1, O_2, \perp, A :: B)(e, v_i, n)$$

Both lemmas are proved by induction on the lexicographical order given by pairs (n, e) (of type $\mathbb{N} \times$ `PVS0Expr`) built with the orders on naturals and (sub)expressions. They are used respectively to prove the next two lemmas, along with the equivalence between predicate ε and function χ.

$$\forall(O_1, O_2, \perp, A, B, e, v_i, v_o) :$$
$$\varepsilon(O_1, O_2, \perp, B)(e, v_i, v_o) \Leftrightarrow \varepsilon(O_1, O_2, \perp, A :: \mathit{map}(\beta(|A|))(B))(\beta(|A|)(e), v_i, v_o)$$

$$\forall(O_1, O_2, \perp, B, v_i, v_o) : \forall(e \mid \mathit{valid\_index\_rec}(e, |A|)) : \forall(A \mid \mathit{valid\_index}(A)) :$$
$$\varepsilon(O_1, O_2, \perp, A)(e, v_i, v_o) \Leftrightarrow \varepsilon(O_1, O_2, \perp, A :: B)(e, v_i, v_o)$$

Formalizing Rice's Theorem also requires a definition of *semantic predicate* of programs and a *Gödelization* of the partial recursive class of PVS0 programs.

The notion of semantic predicate over PVS0 programs is specified as:

$$\mathit{is\_semantic\_predicate}(P) := \forall(\mathit{pvso}_1, \mathit{pvso}_2) :$$
$$(\forall(i, o) : \gamma(\mathit{pvso}_1)(i, o) \Leftrightarrow \gamma(\mathit{pvso}_2)(i, o)) \Rightarrow$$
$$(P(\mathit{pvso}_1) \Leftrightarrow P(\mathit{pvso}_2))$$

For the Gödelization, it is necessary to define a mapping from each PVS0 expression that works with naturals and a mapping from lists of naturals to naturals. The mapping from expressions to naturals uses a bijection from pairs of naturals to naturals, and both are defined as below.

$$\mathit{tuple2nat}(m, n) := \frac{(m + n + 1)(m + n)}{2} + n$$

$$\begin{aligned}
\kappa_e(\mathit{expr}) &:= \texttt{CASES}\ \mathit{expr}\ \texttt{OF} \\
\texttt{vr} &: \ 0; \\
\texttt{cnst}(v) &: \ v \times 5 + 1; \\
\texttt{rec}(j, e_1) &: \ \mathit{tuple2nat}(j, \kappa_e(e_1)) \times 5 + 2; \\
\texttt{op1}(j, e_1) &: \ \mathit{tuple2nat}(j, \kappa_e(e_1)) \times 5 + 3; \\
\texttt{op2}(j, e_1, e_2) &: \ \mathit{tuple2nat}(j, \mathit{tuple2nat}(\kappa_e(e_1), \kappa_e(e_2))) \times 5 + 4; \\
\texttt{ite}(e_1, e_2, e_3) &: \ \mathit{tuple2nat}(\kappa_e(e_1), \mathit{tuple2nat}(\kappa_e(e_2), \kappa_e(e_3))) \times 5 + 5;
\end{aligned}$$

The bijective function from lists of naturals to naturals is called α.

Using the above notions, the injective function from the class of partial recursive functions to naturals is defined as below.

$$\kappa_p(\mathit{pvs}0) = \alpha(\mathit{map}(\kappa_e)(\mathit{pvs}0'4))$$

Finally, the next assumption is required.

$$\forall(E_f) : \exists(\mathit{print} : \texttt{PartialRecursive}) :$$
$$\texttt{LET}\ \ \mathit{self} = (\boldsymbol{O_1}, \boldsymbol{O_2}, \perp, E_f :: \mathit{map}(\beta(|E_f|))(\mathit{print}'4))\ \ \texttt{IN}$$
$$\mathit{partial\_recursive}(\mathit{self}) \ \wedge \ \forall i \ : \varepsilon(\mathit{self})(\beta(|E_f|)(\mathit{print}'4(0)), i, \kappa_p(\mathit{self}))$$

This assumption means that for each PVS0 program list there exists a partial recursive PVS0 program such that they both can be used to build another partial recursive program that outputs the own number of Gödel. As mentioned in the introduction, this result is known as the recursion theorem and, as for universal machines and programs, it is usually given in textbooks in an intuitive manner to convince the reader, but without providing explicit constructions. In Turing complete models, it is possible to design entities that print themselves. From this assumption, depending on the chosen lists of unary and binary functions, if there exists the possibility of creating from a list of PVS0 expressions a partial recursive PVS0 program such that its output for any evaluation is itself, then Rice's theorem holds.

4 Formalization of Rice's Theorem

Rice's theorem, i.e. that any semantic predicate can be decided if and only if it is the set of all PVS0 programs or the empty set, is specified as below.

$$\begin{array}{l}\exists(\mathit{decider} : \texttt{Computable}) : \\ \quad \forall(\mathit{pvso} : \texttt{PartialRecursive}) : \neg(\gamma(\mathit{decider})(\kappa_p(\mathit{pvso}), \bot) \Leftrightarrow P(\mathit{pvso})) \\ \qquad\qquad\qquad\qquad \Leftrightarrow \\ \qquad\qquad\qquad (P = \mathit{fullset} \vee P = \emptyset)\end{array}$$

Proof. Necessity: Suppose that $P = \mathit{fullset}$. Let $\top$ be an element different from $\bot$. The PVS0 program $\mathit{decider} = (\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(\top)])$, decides *fullset*. Now, suppose that $P = \emptyset$. The PVS0 program $\mathit{decider} = (\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(\bot)])$ decides $\emptyset$.
Sufficiency: Proved by contraposition. Let assume that $(P \neq \mathit{fullset} \wedge P \neq \emptyset)$. This implies that there are PVS0 programs, say p and np, such that $P(p)$ and $\neg P(np)$. For reaching a contradiction, suppose that there exists $\mathit{decider} : \texttt{Computable}$ such that:

$$\forall(\mathit{pvso} : \texttt{PartialRecursive})\neg(\gamma(\mathit{decider})(\kappa_p(\mathit{pvso}), \bot) \Leftrightarrow P(\mathit{pvso}))$$

And, consider the program *opp* with kernel:

$$\begin{array}{rl}\mathit{opp} = & [\texttt{ite}(\texttt{rec}(1, \texttt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4| + |p'4|, \texttt{vr})), \\ & \quad \texttt{rec}(1 + |\mathit{decider}'4|, \texttt{vr}), \\ & \quad \texttt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \texttt{vr}))] :: \\ & \mathit{map}(\beta(1))(\mathit{decider}'4) :: \\ & \mathit{map}(\beta(1 + |\mathit{decider}'4|))(\mathit{np}'4) :: \\ & \mathit{map}(\beta(1 + |\mathit{decider}'4| + |\mathit{np}'4|))(p'4)\end{array}$$

Using the assumption that there are programs in the model that can print their own number of Gödel, making $E_f = \mathit{opp}$:

$$\begin{array}{l}\exists(\mathit{print} : \texttt{PartialRecursive}) : \\ \quad \texttt{LET} \quad \mathit{self} = (\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, \mathit{opp} :: \mathit{map}(\beta(\mathit{opp}))(\mathit{print}'4)) \ \texttt{IN} \\ \qquad \mathit{partial\_recursive}(\mathit{self}) \ \wedge \ \forall i \ : \varepsilon(\mathit{self})(\beta(|\mathit{opp}|)(\mathit{print}'4(0)), i, \kappa_p(\mathit{self}))\end{array}$$

To understand how *opp* and *self* work, suppose that for each PVS0 program `PartialRecursive` there is a function with the same name; in pseudo-code:

```
$\mathit{self}(n)$ := IF $\mathit{decider}(\kappa_p(\mathit{self})) \neq \bot$ THEN $\mathit{np}(n)$ ELSE $p(n)$
```

The proof uses Cantor's diagonal argument. If $\mathit{decider}(\kappa_p(\mathit{self})) \neq \bot$, then $P(\mathit{self})$, but *self* behaves as np and thus $\neg P(\mathit{self})$ holds, which is a contradiction. Otherwise, if $\mathit{decider}(\kappa_p(\mathit{self})) = \bot$, then $\neg P(\mathit{self})$, but *self* behaves as p and thus $P(\mathit{self})$ that is a contradiction too. This is the main idea behind the rest of the explanation of the formalization.

The aforementioned assumption implies that there exists an element of the partial recursive class, say *print*, such that:

$$\begin{array}{l}\texttt{LET} \quad \mathit{self} = (\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, \mathit{opp} :: \mathit{map}(\beta(\mathit{opp}))(\mathit{print}'4)) \ \texttt{IN} \\ \quad \mathit{partial\_recursive}(\mathit{self}) \ \wedge \ \forall i \ : \varepsilon(\mathit{self})(\beta(|\mathit{opp}|)(\mathit{print}'4(0)), i, \kappa_p(\mathit{self}))\end{array}$$

Making $pvso = self$ it can be concluded that

$$\neg\gamma(decider)(\kappa_p(self), \bot) \Leftrightarrow P(self)$$

The proof splits into two sub-cases.

Sub-case 1: $\boldsymbol{P(self)}$. In this case, $\neg\gamma(decider)(\kappa_p(self), \bot)$ is concluded.

Since P is a semantic predicate, one has:

$$\begin{aligned}&\forall(pvso_1, pvso_2) :\\ &(\forall(i,o) : \gamma(pvso_1)(i,o) \Leftrightarrow \gamma(pvso_2)(i,o)) \Rightarrow\\ &\qquad (P(pvso_1) \Leftrightarrow P(pvso_2))\end{aligned}$$

Thus, choosing $pvso_1$ as $self$ and $pvso_2$ as np, it gives:

$$(\forall(i,o) : \gamma(self)(i,o) \Leftrightarrow \gamma(np)(i,o)) \Rightarrow (P(self) \Leftrightarrow P(np))$$

Assuming $\forall(i,o) : \gamma(self)(i,o) \Leftrightarrow \gamma(np)(i,o)$, by $P(self)$, $P(np)$ also holds, which is a contradiction since $\neg P(np)$.

Consequently, $\neg\forall(i,o) : \gamma(self)(i,o) \Leftrightarrow \gamma(np)(i,o)$ should hold.

But this is not possible because $self$ performs the same as np as showed below.

Starting by $\gamma(self)(i,o)$ and expanding γ, and from $\varepsilon(self)(self'4(0), i, o)$ replacing $self$ by its definition, one obtains:

$$\varepsilon(self)(opp :: map(\beta(opp))(0), i, o)$$

That by properties of lists and definition of opp gives $\varepsilon(self)(opp(0), i, o)$, and then:

$$\begin{aligned}\varepsilon(self)(&\texttt{ite}(\texttt{rec}(1, \texttt{rec}(1 + |decider'4| + |np'4| + |p'4|, vr)),\\ &\texttt{rec}(1 + |decider'4|, \texttt{vr}),\\ &\texttt{rec}(1 + |decider'4| + |np'4|, \texttt{vr}), i, o)\end{aligned}$$

Then, by the definition of ε and operational semantics of `ite`, one has:

$$\begin{aligned}&\exists\,(v') : \varepsilon(self)(\texttt{rec}(1, \texttt{rec}(1 + |decider'4| + |np'4| + |p'4|, vr)), i, v') \wedge\\ &\texttt{IF } v' \neq \bot \texttt{ THEN } \varepsilon(self)(\texttt{rec}(1 + |decider'4|, \texttt{vr}), i, o)\\ &\texttt{ELSE } \varepsilon(self)(\texttt{rec}(1 + |decider'4| + |np'4|, \texttt{vr}), i, o)\end{aligned}$$

Further, by adequate expansions of predicate ε and application of equalities $self'4(1) = \beta(1)(decider'4(0))$, and $self'4(1 + |decider'4| + |np'4| + |p'4|) = \beta(|opp|)(print'4(0))$, one has:

$$\begin{aligned}&\exists\,(v') : \exists\,(v'') : \exists\,(v''') : i = v''' \wedge\\ &\varepsilon(self)(\beta(|opp|)(print'4(0)), v''', v'') \wedge\\ &\varepsilon(self)(\beta(1)(decider'4(0)), v'', v') \wedge\\ &\texttt{IF } v' \neq \bot \texttt{ THEN } \varepsilon(self)(\texttt{rec}(1 + |decider'4|, \texttt{vr}), i, o)\\ &\texttt{ELSE } \varepsilon(self)(\texttt{rec}(1 + |decider'4| + |np'4|, \texttt{vr}), i, o)\end{aligned}$$

And then, by Skolemization of the existentially quantified variables one has:

$$\begin{aligned}&i = v''' \wedge\\ &\varepsilon(self)(\beta(|opp|)(print'4(0)), v''', v'') \wedge\\ &\varepsilon(self)(\beta(1)(decider'4(0)), v'', v') \wedge\\ &\texttt{IF } v' \neq \bot \texttt{ THEN } \varepsilon(self)(\texttt{rec}(1 + |decider'4|, \texttt{vr}), i, o)\\ &\texttt{ELSE } \varepsilon(self)(\texttt{rec}(1 + |decider'4| + |np'4|, \texttt{vr}), i, o)\end{aligned}$$

By the assumption, $\forall i\ : \varepsilon(\mathit{self})(\beta(|\mathit{opp}|)(\mathit{print}'4(0)), i, \kappa_p(\mathit{self}))$, and instantiating $i = v'''$ one obtains:

$$\begin{aligned}
&\varepsilon(\mathit{self})(\beta(|\mathit{opp}|)(\mathit{print}'4(0)), v''', \kappa_p(\mathit{self})) \wedge\\
&\varepsilon(\mathit{self})(\beta(|\mathit{opp}|)(\mathit{print}'4(0)), v''', v'') \wedge\\
&\varepsilon(\mathit{self})(\beta(1)(\mathit{decider}'4(0)), v'', v') \wedge\\
&\texttt{IF}\ v' \neq \bot\ \texttt{THEN}\ \varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4|, \texttt{vr}), i, o)\\
&\texttt{ELSE}\ \varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \texttt{vr}), i, o)
\end{aligned}$$

Since the relation ε (is formalized to be) functional, one has that $v'' = \kappa_p(\mathit{self})$. Thus, the expression below should be considered.

$$\begin{aligned}
&\varepsilon(\mathit{self})(\beta(1)(\mathit{decider}'4(0)), \kappa_p(\mathit{self}), v') \wedge\\
&\texttt{IF}\ v' \neq \bot\ \texttt{THEN}\ \varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4|, \texttt{vr}), i, o)\\
&\texttt{ELSE}\ \varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \texttt{vr}), i, o)
\end{aligned}$$

By using the lemma of code shift,

$$\varepsilon(\mathit{self})(\beta(1)(\mathit{decider}'4(0)), \kappa_p(\mathit{self}), v') \Leftrightarrow \varepsilon(\mathit{decider})(\mathit{decider}'4(0), \kappa_p(\mathit{self}), v')$$

Thus one obtains,

$$\begin{aligned}
&\varepsilon(\mathit{decider})(\mathit{decider}'4(0), \kappa_p(\mathit{self}), v') \wedge\\
&\texttt{IF}\ v' \neq \bot\ \texttt{THEN}\ \varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4|, \texttt{vr}), i, o_1)\\
&\texttt{ELSE}\ \varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \texttt{vr}), i, o)
\end{aligned}$$

By the hypothesis of this case, one has $\neg\gamma(\mathit{decider})(\kappa_p(\mathit{self}), \bot)$ that means that $v' \neq \bot$. Thus,

$$\varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4|, \texttt{vr}), i, o)$$

By adequate expansions of predicate ε, Skolemization of the obtained existentially quantified variable as v'_1 and replacing the necessary variables, one obtains:

$$\varepsilon(\mathit{self})(\mathit{np}'4(0), i, o)$$

Applying lemmas of code shift:

$$\varepsilon(\mathit{np})(\mathit{np}'4(0), i, o)$$

That is equivalent to $\gamma(\mathit{np})(i, o)$. Thus $\neg\forall(i, o) : \gamma(\mathit{self})(i, o) \Leftrightarrow \gamma(\mathit{np})(i, o)$ does not hold and that is a contradiction.
Sub-case 2: $\neg \boldsymbol{P(self)}$. It follows analogously to sub-case 1, except for the supposition that P is a semantic predicate where pvso_1 and pvso_2 are instantiated respectively as *self* and p, which gives a contradiction.

□

5 Applications

Generality of Rice's Theorem allows simple formalizations of important undecidability results in computability theory. In particular, since our proof does not depend on the undecidability of the Halting Problem, we obtain it as a direct consequence.

Corollary 5.1 (Undecidability of the Halting Problem)

$$\begin{aligned}
&\neg\exists(\mathit{decider} : \texttt{Computable}) :\\
&\qquad\forall(\mathit{pvso} : \texttt{PartialRecursive}) : \neg(\gamma(\mathit{decider})(\kappa_p(\mathit{pvso}), \bot) \Leftrightarrow T_\varepsilon(\mathit{pvso}))
\end{aligned}$$

Proof. The formalization uses Rice's Theorem instantiating the semantic predicate as T_ε. Since the set T_ε is neither equal to the empty set nor to the whole set `PartialRecursive`, there exists no computable decider for this set. To prove this, it is shown that the `PartialRecursive` constant program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\mathtt{cnst}(0)])$ belongs to T_ε, while a simple loop `PartialRecursive` program specified as $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\mathtt{rec}(0, \mathtt{vr})])$ does not.

For the loop above, notice that the input of the recursive call does not change, so that the execution of the program will repeat the recursive call infinitely. □

The PVS theory that complements this paper includes both the formalization of the corollary above and a direct formalization of the undecidability of the Halting Problem for the multiple-function PVS0 model in the spirit of [5].

Corollary 5.2 (Undecidability of Existence of Fixed Points)

$$\begin{array}{l} \neg\exists(\mathit{decider} : \mathtt{Computable}) : \\ \quad \forall(\mathit{pvso} : \mathtt{PartialRecursive}) : \neg(\gamma(\mathit{decider})(\kappa_p(\mathit{pvso}), \bot) \Leftrightarrow \exists(p) : \gamma(\mathit{pvso})(p, p)) \end{array}$$

Proof.
The formalization instantiates Rice's Theorem using the semantic predicate

$$\lambda(\mathit{pvso} : \mathtt{PartialRecursive}) : \exists(p) : \gamma(\mathit{pvso})(p, p)$$

The predicate is then shown to be different from the empty set and from the whole set `PartialRecursive`. Indeed, on one side, the predicate holds for the program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\mathtt{cnst}(0)])$, showing that it is different from empty set. On the other side, it does not hold for the program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\mathtt{op2}(i, \mathtt{vr}, \mathtt{cnst}(1))])$ that performs the same as $\lambda(n : \mathbb{N}) : \mathit{tuple2nat}(n, 1)$, concluding that the predicate is not equal to `PartialRecursive`. □

Corollary 5.3 (Undecidability of Self Replication)

$$\begin{array}{l} \neg\exists(\mathit{decider} : \mathtt{Computable}) : \\ \quad \forall(\mathit{pvso} : \mathtt{PartialRecursive}) : \\ \quad\quad \neg\ (\ \ \gamma(\mathit{decider})(\kappa_p(\mathit{pvso}), \bot) \ \ \Leftrightarrow \\ \quad\quad\quad\quad \exists(p : \mathtt{PartialRecursive}) : \forall(i) : \gamma(p)(i, \kappa_p(p)) \wedge \gamma(\mathit{pvso})(i, \kappa_p(p)) \ \ ) \end{array}$$

Proof.
To formalize it, it is necessary to instantiate the predicate in the Rice's theorem as

$$\lambda(\mathit{pvso} : \mathtt{PartialRecursive}) : \exists(p : \mathtt{PartialRecursive}) : \forall(i) : \gamma(p)(i, \kappa_p(p)) \wedge \gamma(\mathit{pvso})(i, \kappa_p(p))$$

The next step is showing that the predicate is neither the empty set nor the full `PartialRecursive` set. Using the assumption of the recursion theorem and instantiating it with $[\mathtt{rec}(1, \mathtt{vr})]$, one shows that the predicate is not empty. On the other side, the program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\mathtt{op2}(i, \mathtt{cnst}(1), \mathtt{vr})])$ shows that the predicate is not the whole `PartialRecursive` set.
□

Corollary 5.4 (Undecidability of Functional Equivalence)

$$\begin{array}{l} \neg\exists(\mathit{decider} : \mathtt{Computable}) : \\ \quad \forall(\mathit{pvso}_0, \mathit{pvso}_1 : \mathtt{PartialRecursive}) : \\ \quad\quad \neg\ (\ \ \gamma(\mathit{decider})(\mathit{tuple2nat}(\kappa_p(\mathit{pvso}_0), \kappa_p(\mathit{pvso}_1)), \bot) \Leftrightarrow \\ \quad\quad\quad\quad\quad\quad (\ \forall(i, o) : \gamma(\mathit{pvso}_0)(i, o) \Leftrightarrow \gamma(\mathit{pvso}_1)(i, o) \ ) \ \ ) \end{array}$$

Proof.
Suppose that there exists a `Computable` program *decider* that decides the above equivalence between functions. Then, instantiate pvso_0 above as the constant zero program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\mathtt{cnst}(0)])$. Then, the problem functional equivalence problem is reduced to decide if a program preforms the same as the constant zero program. The next step is instantiating the Rice's Theorem with the following semantic predicate:

$$\lambda(pvso : \texttt{PartialRecursive}) : \forall(i,o) : \gamma(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(0)])(i,o) \Leftrightarrow \gamma(pvso)(i,o)$$

To show that the predicate is neither the empty not the full `PartialRecursive` set, it is enough to prove that the constant zero program belongs to the predicate and that the constant one program does not. After that, one uses the assumed program *decider* to build another program for the equivalence with the constant zero program; this program is build as

$$(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{rec}(1, \texttt{op2}(i, \kappa_p(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(0)]), vr))] :: decider`4),$$

where i is the index to the *tuple2nat* function. Using the shifting code lemmas, it can indeed be simplified to decide equivalence to the constant zero program.

This formalization, requires also proving that the program built above is in fact `Computable`. This is a consequence of *decider* being assumed as a `Computable` program and then being terminating too. The proof concludes by applying again the shifting code lemmas and to show that the program built above is also terminating. □

6 Related Work

Nowadays, mechanical proofs of computability properties is not only of interest as an exercise of formalization, but also of great importance to provide formal support for computational models used for pragmatical issues. As mentioned in the introduction, the main aim of the PVS0 computational single- and multiple-function models is related with the development of automation mechanisms to verify termination of PVS programs. In [5], PVS0 programs not only consist of a single function, but also were constrained in inductive levels such that in the level zero only the basic functions successor, greater than and projections are allowed and, in subsequent levels, other `Computable` functions can be specified allowing calls to functions of the previous level as operators. Building composition of such PVS0 programs was not straightforward, which makes difficult the formalization of results such as Turing completeness and Rice's Theorem. As seen in Section 2 on semantics of PVS0, the composition of programs specified in the multiple-function PVS0 language is straightforwardly achieved by application of the operator β.

For the single-function PVS0 language, the composition of two (non necessarily terminating) programs requires the construction of a third program which cannot be specified in a general manner since this depends on the (combinatorial) structure of the input programs. In the proof of undecidability of the Halting Problem in [5], this problem was easily resolved since only very particular composition constructions (of assumed terminating functions) were necessary. Such difficulties are solved in the current work using as model a language which supports specification of programs that consist of several functions that can call not only themselves recursively, but that can also call each other.

The equivalence between termination criteria was formalized for the single-function PVS0 model considered in [5]. However, it was not formalized completely for the current multiple-function PVS0 model. Theoretically, all termination criteria mentioned in the introduction (references [13], [14], [3], [17], [2]) work for both models, but technically, some of the criteria require a re-adaptation to deal simultaneously with static analysis of multiple-function programs that allow even mutual recursion (which, in particular, is avoided in the PVS functional specification language).

Computability properties have been formalized since the development of the first theorem provers and proof assistants. As well-known examples one can mention the mechanical proof of the undecidability of the Halting Problem in [4] using as model of computation the LISP language, as well as the formalization in Agda of the same theorem in [10] using as a model of computation axioms over the elements of an abstract type Prog.

Here, the focus is restricted to recent works in which computability results have been formalized over such computational models related with lambda calculus and programming languages. In [9], Foster and Smolka used as model of computation call-by-value lambda calculus, which is a Turing Complete model of computation, where beta-reduction can be applied whenever the beta-redex is not below an abstraction and the argument is an abstraction. For this model, the authors formalized several computational properties including Rice's Theorem; indeed, they formalized that semantic predicates such that there are elements both in them and in their complements, are not recognizable. Using this result they concluded Rice's Theorem. Also, Norrish formalized in [15], using HOL4, Rice's Theorem for the model of lambda calculus, among others properties such as the existence of universal machines and an instance of the s-m-n theorem.

In addition to Rice's Theorem, there are another computability results formalized over linguistic computational models. Forster, Kirsk and Smolka formalized in [7] undecidability of validity, satisfiability, and provability of first-order formulas following a synthetic approach based on the computation native to Coq's constructive type theory. Forster and Larchey-Wendling formalized in [8] using Coq, the reduction of the Post Correspondence Problem via binary stack machines and Minsky machines to provability of intuitionistic linear logic. They started with Post Correspondence Problem and built a chain of reductions passing through binary

Post Correspondence Problem, binary Post Correspondence Problem with indices, binary stack machines, Minsky machines and finally provability of intuitionistic linear logic. Also, Larchey-Wendling formalized in [11] that any function of the type $\mathbb{N}^k \to \mathbb{N}$ specified using Coq is total. Recently, previous author together with Forest formalized in [12] the undecidability of Hilbert's Tenth Problem using a chain of reductions of problems: Halting Problem for TMs, Post's Correspondence Problem, a specialized Halting Problem for Minsky Machines, FRACTAN (a language model that deals with register machines) termination, and solvability of Diophantine logic and of Diophantine equations.

7 Conclusions, Work in Progress and Future Work

A formalization of Rice's Theorem in PVS is given over the functional language PVS0 seen as a model of computation. The formalization uses a Gödelization of PVS0 programs and follows Cantor's diagonal argument to build a contradiction of the existence of a PVS0 program that can decide semantic predicates over PVS0 programs. Applications of this formalization include formalizations of corollaries for undecidability of the Halting Problem, functional equivalence problem, existence of fixed points problem and self-replication.

Other results of interest to be formalized for the PVS0 language model are the s-m-n theorem, the undecidability of Post Correspondence Problem, Post's Theorem, Turing-completeness, the existence of a universal machine, the existence of self-replicate machines, linear speedup theorem, tape compression theorem, time hierarchy theorem, space hierarchy theorem, etc. The main difficulty, but also the interesting aspect of such formal developments in PVS0, is that the classical proofs of these theorems are performed over specific models such as lambda-calculus and Turing-machines. Even more interesting will be the formalization of other undecidability results outside the context of properties of computational models such as the Word Problem for algebraic structures ([16]) and Hilbert's Tenth Problem ([12]).

Recent examples of related developments include the formalizations in Coq of the Post's theorem for weak call-by-value lambda-calculus [9] and of the undecidability of the Post Correspondence Problem via reduction of the Halting Problem for Turing machines [6]. Despite the existence of correspondences between the functional model PVS0, lambda-calculus and Turing machines, which may be explored for the formalization of such theorems for PVS0 programs, obvious difficulties are that these formalizations are strongly related to the respective computational model and that (formally building) the required translations is not straightforward.

Since in the PVS0 library, equivalence results for termination criteria were formalized for the single-function model, it is also of great interest formalizing these results for the multiple-function model. There are different manners to achieve this objective such as: ensuring Turing Completeness for both models, and thus equivalence of the models; formalizing directly the equivalence between the models and building single-function to and from multiple-function program translations; or by directly adapting the results for the multiple-function model.

Related current work includes the formalization of the fixed-point theorem for the multiple-function PVS0 model. The proof of this theorem involved the assumption of the existence of a universal program (as mentioned before, a standard assumption in textbooks, but required to be constructed in complete formalizations), which was required to build a diagonal program that performs analogously to the lambda term $\lambda xy \cdot (xx)y$.

References

[1] Thomas Arts. Termination by Absence of Infinite Chains of Dependency Pairs. In *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Linköping, 1996, Proceedings*, 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. *Theoretical Computer Science*, 236:133–178, 2000.

[3] Andréia Borges Avelar. *Formalização da Automação da Terminação Através de Grafos com Matrizes de Medida*. PhD thesis, Universidade de Brasília, Departamento de Matemática, Brasília, Distrito Federal, Brazil, 2015.

[4] Robert Stephen Boyer and J Strother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the Association for Computing Machinery*, 31(3):441–458, 1984.

[5] Thiago Mendonça Ferreira Ramos, César Augusto Muñoz, Mauricio Ayala-Rincón, Mariano Miguel Moscato, Aaron Dutle, and Anthony Narkawicz. Formalization of the Undecidability of the Halting Problem for a Functional Language. In *Logic, Language, Information, and Computation - 25th International Workshop, WoLLIC 2018*, volume 10944 of *Lecture Notes in Computer Science*, pages 196–209. Springer, 2018.

[6] Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-Related Computational Reductions in Coq. In *Interactive Theorem Proving - 9th International Conference, ITP 2018*, volume 10895 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2018.

[7] Yannick Forster, Dominik Kirst, and Gert Smolka. On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pages 38–51. ACM, 2019.

[8] Yannick Forster and Dominique Larchey-Wendling. Certified Undecidability of Intuitionistic Linear Logic via Binary Stack Machines and Minsky Machines. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pages 104–117. ACM, 2019.

[9] Yannick Forster and Gert Smolka. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *Interactive Theorem Proving - 8th International Conference, ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017.

[10] Kristofer Johannisson. Formalizing the halting problem in a constructive type theory. In *Types for Proofs and Programs, International Workshop, TYPES 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 2000.

[11] Dominique Larchey-Wendling. Typing Total Recursive Functions in Coq. In *Interactive Theorem Proving - 8th International Conference, ITP 2017*, volume 10499 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2017.

[12] Dominique Larchey-Wendling and Yannick Forster. Hilbert's Tenth Problem in Coq. In *Proceedings FSCD 2019*, 2019. To appear. Available at `https://uds-psl.github.io/H10`, Saarland University.

[13] Chin Soon Lee, Neil Deaton 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.

[14] Panagiotis Manolios and Daron Vroon. Termination Analysis with Calling Context Graphs. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2006.

[15] Michael Norrish. Mechanised Computability Theory. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.

[16] Emil L. Post. Recursive unsolvability of a problem of Thue. *The Journal of Symbolic Logic*, 12(1):1–11, 1947.

[17] Alan Mathison Turing. Checking a Large Routine. In Martin Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.