

Formalization of the Computational Theory of a Turing Complete Functional Language Model

Thiago Mendonça Ferreira Ramos ·
Ariane Alves Almeida ·
Mauricio Ayala-Rincón

Received: date / Accepted: date

Abstract This work presents a formalization in PVS of the computational theory for a computational model given as a class of partial recursive functions called **PVS0**. The model is built over basic operators, which, when restricted to constants, successor, projections, greater-than, and bijections from tuples of naturals to naturals, results in a proven (formalized) Turing complete model. Complete formalizations of the Recursion Theorem and Rice's Theorem are discussed in detail. Other relevant results, such as the Halting Problem's undecidability and the Fixed-Point Theorem, were also fully formalized.

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

1 Introduction

It is well-known that all Turing complete models of computation have some expressiveness limits. Thus, not all kinds of *problems* can be solved, and *programs* over each of these models with the same or different semantics cannot be distinguished. One of these limits, for instance, is given by Rice's Theorem, which 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.

Formalizations of undecidability results developed in the proof assistant PVS are presented. These results are proved over a computation model called **PVS0**, a variant of a first-order functional language. The language includes 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 correctness of mechanisms to automate verification of termination in PVS, which makes of high interest the exploration of the computability theory of **PVS0** as

Thiago Mendonça Ferreira Ramos[†], Ariane Alves Almeida[†], Mauricio Ayala-Rincón^{†,‡}
Departments of [†]Computer Science and [‡]Mathematics, Universidade de Brasília
E-mail: ayala@unb.br

a model of computation. Indeed, `PVS0` is applied (i.e. used as a model of computation) to formalize the equivalence among different criteria of termination, including size change principle ([20]) based termination criteria such as Manolios and Vroom’s Calling Context Graphs [21] and Avelar *et al* Matrix Weighted Graphs [4], as well as Turing termination [28] (that is indeed the criterion applied for the PVS specification language), and Dependency Pairs termination (see e.g., [2][3][1]). The libraries for `PVS0`, which include equivalence proofs of these termination criteria, are available as part of the NASA LaRC PVS library at <https://github.com/nasa/pvslib/tree/master/PVS0>.

This paper aims to formalize the *computational theory* of `PVS0` as a computation model that is a second relevant objective to understand the model. In addition to verification of termination criteria over `PVS0`, it is essential 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 Halting Problem’s undecidability for the `PVS0` model [7]. 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. `PVS0` is a more realistic model, accepting, similarly to functional specifications, a list of functions such that each function can call and be called by each other in the list using their indices. This model is called the *multiple-function* `PVS0` model (or just `PVS0` when no confusion arises). The undecidability of the Halting Problem was fully and directly formalized for the multiple-function model too. However, this result can also be obtained just as a corollary of Rice’s theorem’s formalization.

The language `PVS0` was designed to be similar to the specification language of PVS to allow a meta-theoretical study of the properties associated with PVS itself. The results obtained for the language `PVS0` imply, for instance, that PVS itself is Turing Complete and, under specific restrictions, satisfies all computational properties formalized in this development. Therefore, this study attempts not only a better understanding but also eventual improvements to the PVS model. An essential difference between PVS and the `PVS0` model is that the input and output types of a `PVS0` program must be the same since the model works with a unique input/output type, while this is not required in PVS. Other differences are that `PVS0` allows mutual recursion, while PVS does not; and, that the PVS grammar is much richer than the `PVS0` grammar. The choice of a *minimal* `PVS0` grammar aims to simplify formalizations; indeed, reducing the number of grammatical elements also reduces the number of cases to be considered in proofs of properties of the `PVS0` model. Additionally, having a unique input/output type facilitates the specification in PVS of the syntax and operational semantics of `PVS0`. Nevertheless, the `PVS0` grammar is rich enough to implement any PVS function.

The unique input/output type of the `PVS0` language model is passed as a parameter of the PVS development that for the formalized theorems is set as the type of naturals. The `PVS0` theory also requires parameters: lists of basic operators (PVS functions) and an element of the input/output type to interpret as false. Keeping these parameters fixed defines a class of partial recursive functions. Basic operators, including successor, greater-than, and projections, provide a model formalized to be Turing Complete. However, the model may also specify non-computable functions when basic operators that are non-computable PVS functions are allowed.

The main contributions of this work are formalizations of the following properties of the multiple-function PVS0 model.

- Turing Completeness. The formalization proves that the class of partial recursive PVS0 programs, built from basic functions and predicates (projections, successor, constants, greater-than), are closed under the operations of composition, minimization, and primitive recursion. It follows the lines of proofs such as the one in [27] that shows λ -definability of partial recursive functions. For the formalization of this result, some specialized constructions were necessary. For instance, for composition and primitive recursion, since a PVS0 program should receive as argument a natural that represents a tuple of naturals resulting from applications of several PVS0 programs, it was necessary to construct bijections from tuples of naturals to naturals.
- Recursion Theorem. This is the most elaborate of all proved theorems. The formalization is similar to programming a computer virus using a functional language with one difference: the idea is processing the own Gödel number, as a quine does, of a partial recursive PVS0 program instead of the own code. The proof uses a bijective Gödelization of partial recursive PVS0 programs; however, bijectivity is not required for the Recursion Theorem as it is for the Fixed-Point Theorem. The Gödelization was implemented based on a Gödelization of PVS0 expressions, and each PVS0 program is mapped into a natural that encodes the tuple of naturals associated with its expressions. This construction avoids implementing an elaborated PVS0 program that calculates its Gödel number. The formalization follows the lines of proof as given in [24].
- Rice’s Theorem. It was formalized as a corollary of the Recursion Theorem used to build a partial recursive PVS0 program, which processes its Gödel number. If it is the number of a program that satisfies any semantic property, then the program behaves as if it does not satisfy the property; otherwise, it behaves as if it satisfies it. This formalization follows the classical diagonalization argumentation as done in [24] for Turing Machines.
- Additional results such as the Halting Problem’s undecidability and the Fixed-Point Theorem were also formalized. There are two versions of the Theorem of the Halting Problem’s undecidability. One says that it is undecidable if a program halts for a specific input (Halting Problem). Another one says that it is undecidable if a program halts for all inputs (Uniform Halting Problem). The latter was proved just as a corollary of the Rice’s Theorem. The former was proved using diagonalization and arbitrary Gödelizations of partial recursive PVS0 programs, and a bijection from tuples of naturals to naturals to encode PVS0 programs and inputs. The formalization follows the proof style in [24] for Turing Machines. The Fixed-Point Theorem was formalized as a consequence of the fact that it is possible to build the universal PVS0 program. Besides, it uses a *diagonal* program whose semantics is receiving two arguments: the first one is a program that transforms an input program into another one, and the second one is a value. The *diagonal* program applies the first argument to itself and the second argument. This proof is the only one in the development that uses the bijectivity of the Gödelization of partial recursive PVS0 programs. The construction follows the proof in [8].

The PVS development of the computability theory for the multiple-function PVS0 model has its hierarchy synthesized in Figure 1.

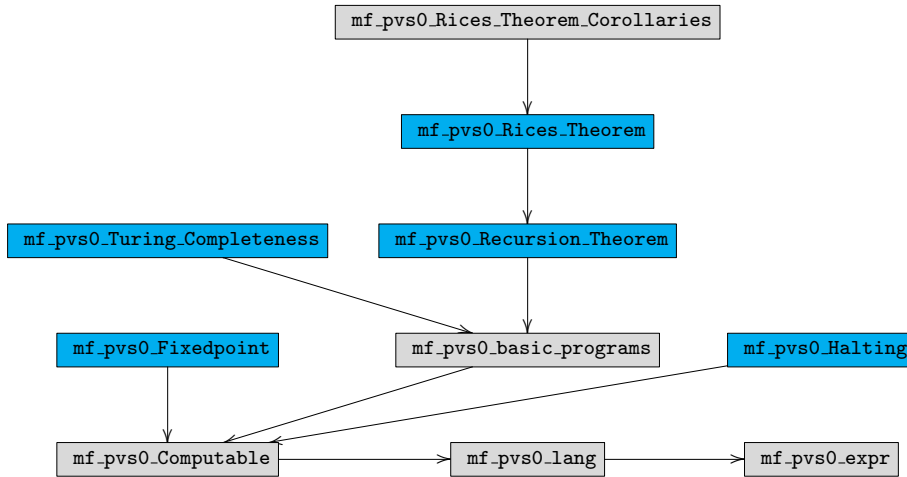


Fig. 1 Hierarchy of the PVS0 theory related with this extension

To prove the Recursion Theorem, a partial recursive PVS0 program that computes its Gödel number is built. The construction is achieved by dividing the program list of expressions into three parts. The first part is used to build a number related to the Gödel number that corresponds to a combination of the second and the third parts. The second part uses the result of the first one to calculate the Gödel number of the given program. The third part is used to process the result of the second one. The formalization is based on the proof presented in Sipser’s textbook ([24]) for Turing Machines, where Turing Machines are used to print and process their descriptions. In this work, Sipser’s approach is adapted to build functional programs that output their Gödel number. The main difficulty is that Sipser’s proof informally describes how to build such Turing Machines, while here, to have a complete formalization, concrete constructions of such partial recursive PVS0 program are, of course, required.

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 the separability of machines’ semantic properties. The main differences between classical proofs and the one given in this work are that in addition to work with a functional programming model, the proof does not depend on the Halting Problem’s undecidability, being concluded from the Recursion Theorem without using any translation to or from other computational models. Sipser’s textbook [24] includes proof that is similar to the one given here but providing several informal descriptions and justifications, which are not possible in the current formalization. In particular, a crucial difference is that here, the Gödelization is explicitly built.

Straightforward formalizations of the Uniform Halting Problem’s undecidability, functional equivalence problem, the existence of fixed points problem, and self-replication problem are obtained as corollaries of Rice’s Theorem.

In textbooks, assumptions such as the existence of universal machines and programs, the existence of bijections between inputs, machines, and programs,

and naturals, as well as Gödelization of machines and programs are intuitively given without providing complete constructions. Even assumptions such as the Recursion Theorem and Turing Completeness are accepted without constructive proofs. To rule out such kinds of assumptions and get complete proofs, specifically related constructions were fully formalized.

Organization: After giving the semantics of the PVS0 model and the specifications of computable and partial recursive classes in Sections 2 and 3 (related with files `mf_pvs0_expr`, `mf_pvs0_lang` and `mf_pvs0_computable` in Figure 1), Sections 4, 5 and 6 explain respectively the formalizations of Turing Completeness, Recursion Theorem and of Rice’s Theorem (files `mf_pvs0_Turing_Completeness`, `mf_pvs0_RecursionTheorem` and `mf_pvs0_Rices_Theorem`). Section 6 also discusses crucial results formalized as simple corollaries of Rice’s Theorem including the undecidability of the Uniform Halting Problem (file `mf_pvs0_Rices_Theorem_Corollaries`). Then, before concluding and discussing current and future work in Section 8, Section 7 discusses another important theorems included in the development (files `mf_pvs0_Fixedpoint` and `mf_pvs0_Halting`) and related work. The formalization requires the installation of the (PVS NASA library [🔗](#)) and is provided as part of this submission, directly in the (PVS0 theory [🔗](#)) that belongs to this library. The paper uses the symbol [🔗](#) as hyperlinks to specific files between the PVS0 theory.

2 Semantics of PVS0 Programs

Expressions of the PVS0 functional language have the grammar below.

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

Above, T is an uninterpreted type over which PVS0 expressions are interpreted. The grammar is implemented as an abstract datatype built from T in the formalization. This type is the type of input/output used to evaluate expressions (and programs). Constants of type T are represented by the constructor `cnst(T)` and the symbol `vr` is the unique symbol of the variable. The main advantage of using such simple grammar is generating a low number of cases to be analyzed in the proofs. The grammar is compact but expressive enough, allowing other functional language structures as “let-in”, “case-of”, and “match-with”, among others. will increment the proof sizes. `op1` and `op2` denote respectively unary and binary built-in operators indexed by naturals. These indices reference positions in lists of unary and binary PVS functions used to interpret operator symbols. The symbol `rec` is for function calls and uses natural indices required to select the function to be called in a PVS0 program. In the evaluation, the selected function is applied 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-then-else.

From the specification of this grammar as an abstract datatype, PVS generates the required basic functions and axioms; for instance, the *subterm* relation is generated as below.

$$\begin{aligned}
& \text{subterm}(x, y) := x = y \vee \\
& \quad \text{CASES } y \text{ OF} \\
& \quad \quad \text{vr} : \text{False}; \\
& \quad \quad \text{cnst}(v) : \text{False}; \\
& \quad \quad \text{rec}(j, e_1) : \text{subterm}(x, e_1); \\
& \quad \quad \text{op1}(j, e_1) : \text{subterm}(x, e_1); \\
& \quad \quad \text{op2}(j, e_1, e_2) : \text{subterm}(x, e_1) \vee \text{subterm}(x, e_2); \\
& \quad \text{ite}(e_1, e_2, e_3) : \text{subterm}(x, e_1) \vee \text{subterm}(x, e_2) \vee \\
& \quad \quad \text{subterm}(x, e_3);
\end{aligned}$$

The kernel of a PVS0 program is a non-empty list of PVS0 expressions whose main expression (the first expression to be evaluated) is the one at the head of the list. Recursive calls, $\text{rec}(i, _)$, are interpreted as evaluations (or function calls) of the i^{th} PVS0 expression in the list. PVS0 programs also include lists of unary and binary functions to interpret the symbols of unary and binary operators, and an element of the input/output type of the programs 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, \perp, E_f) , where O_1 and O_2 are the lists of unary and binary functions (specified in PVS), \perp is an element of T , and E_f is the program's kernel. PVS supplies other alternative data structures instead of lists such as finite sequences, sets, and functions that can be used to specify the elements related to PVS0 programs. The choice of lists is justified because of their simplicity and that PVS provides a complete game of operators verified over lists. Note that the evaluation of the expression given as the first argument of an `ite` symbol must also be an element of type T interpreted as a Boolean. Thus, to guarantee that `ite` has the semantics of if-then-else instructions, the interpretation of false as a specific element \perp of type T is necessary. Built-in operators (O_1 and O_2) are necessary because they allow an adaptation of a PVS0 program to represent any PVS function, verifying such operators separately. The Recursion and Rice's Theorems are formalized, constraining basic operators only to have computable functions. However, it is relevant to stress that PVS also allows the specification of non-computable functions providing to PVS0, in this manner, the required flexibility to formalize results about oracle programs.

List of n -elements are denoted as $[a_0, \dots, a_{n-1}]$. $L(i)$ denotes the i^{th} element of the list L , and $|L|$ its length. The tail of a non-empty list L is denoted as $\text{cdr}(L)$, and $L_1 :: L_2$ denotes the concatenation of the lists L_1 and L_2 . The mapping of the list L using the function f is denoted as $\text{map}(f)(L)$.

The (eager) evaluation predicate ε (specified as `semantic_rel_expr` [↗](#) in the PVS0 theory), for PVS0 programs is defined in Table 1.

The predicate ε is specified inductively. Specifying this predicate as a recursive Boolean is impossible since there is no termination measure for semantic evaluation. This choice enables us to correctly specify the notion of semantic evaluation without providing a termination measure. Besides that, when the predicate runs an infinite loop, it means that the evaluated PVS0 program does not answer an output.

Note that the predicate ε is curried. Currying the first four parameters of ε , which correspond to a PVS0 program, is convenient to deal with the own evaluation relation associated to each PVS0 program; thus, the relation $\varepsilon(\text{pvs}_0)$ is the

evaluation relation for the specific program pvs_0 . PVS currying feasibility is applied to several predicates and functions related to operational properties of PVS0 programs.

Table 1 PVS0 program evaluation predicate - $pvs_0 = (O_1, O_2, \perp, E_f)$ ([semantic_rel_expr](#))

$$\begin{aligned}
& \varepsilon(pvs_0)(e, v_i, v_o) := \text{CASES } e \text{ OF} \\
\text{cnst}(v) & : v_o = v; \\
\text{vr} & : v_o = v_i; \\
\text{op1}(j, e_1) & : \exists (v' : T) : \\
& \quad \varepsilon(pvs_0)(e_1, v_i, v') \wedge \\
& \quad \text{IF } j < |O_1| \text{ THEN } v_o = O_1(j)(v'); \\
& \quad \text{ELSE } v_o = \perp \\
\text{op2}(j, e_1, e_2) & : \exists (v', v'' : T) : \\
& \quad \varepsilon(pvs_0)(e_1, v_i, v') \wedge \\
& \quad \varepsilon(pvs_0)(e_2, v_i, v'') \wedge \\
& \quad \text{IF } j < |O_2| \text{ THEN } v_o = O_2(j)(v', v''); \\
& \quad \text{ELSE } v_o = \perp \\
\text{rec}(j, e_1) & : \exists (v' : T) : \varepsilon(pvs_0)(e_1, v_i, v') \wedge \\
& \quad \text{IF } j < |E_f| \text{ THEN} \\
& \quad \quad \varepsilon(pvs_0)(E_f(j), v', v_o) \\
& \quad \text{ELSE } v_o = \perp \\
\text{ite}(e_1, e_2, e_3) & : \exists (v' : T) : \varepsilon(pvs_0)(e_1, v_i, v') \wedge \\
& \quad \text{IF } v' \neq \perp \text{ THEN } \varepsilon(pvs_0)(e_2, v_i, v_o) \\
& \quad \text{ELSE } \varepsilon(pvs_0)(e_3, v_i, v_o).
\end{aligned}$$

Parameters v_i and v_o of the predicate ε are the input and output values, and the parameter e is the (sub)expression being evaluated of the kernel of the PVS0 program (O_1, O_2, \perp, E_f) , for short denoted as pvs_0 . When evaluating this expression, its subexpressions can match expressions that perform operations, such as the built-in operators or (recursive) function calls. In these cases, the index j is used either to select the desired operator in the lists of given unary and binary operators, O_1 and O_2 or the selected function in the list of expressions E_f of the program pvs_0 .

Using the predicate ε , another predicate is defined that holds for PVS0 programs and correct inputs and outputs, specified as below, where $pvs_0'4$ denotes the projection of the fourth element of the 4-tuple pvs_0 .

$$\gamma(pvs_0)(v_i, v_o) := \varepsilon(pvs_0)(pvs_0'4(0), v_i, v_o)$$

Note that γ ([semantic_rel](#)) starts the evaluation from the main function of the program that is $pvs_0'4(0)$, the head of the list of expressions $pvs_0'4$, the same as E_f .

Aiming to prove properties related to automation of termination, semantic termination of PVS0 programs was specified in [7]. Semantic termination is required to prove completeness and equivalence of practical termination criteria and to

formalize computability results such as undecidability of the Halting Problem and Rice's theorem. The *semantic termination predicate* is specified as below.

$$T_\varepsilon(pvs_0, v_i) := \exists (v_o : T) : \gamma(pvs_0)(v_i, v_o).$$

This predicate T_ε (**determined?** [↗](#)) states that for a given program pvs_0 and input v_i , the evaluation terminates with the output value v_o . The program pvs_0 is *total with respect to ε* if it satisfies the predicate $T_\varepsilon(pvs_0)$ (**terminating?** [↗](#)) specified as below.

$$T_\varepsilon(pvs_0) := \forall (v : T) : T_\varepsilon(pvs_0, v).$$

Also, the predicate is deterministic (**deterministic** [↗](#)), specified as below.

$$\forall (v_i, v_o, v'_o) : \gamma(pvs_0)(v_i, v_o) \wedge \gamma(pvs_0)(v_i, v'_o) \Rightarrow v_o = v'_o$$

Both the formalizations of the Halting Problem's undecidability and Rice's Theorem (as given in [7] and in this paper, respectively) require building the composition of functions and programs, which is used to specify 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 [7], it was not the case. The required compositions in [7] 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 sharing the same unary and binary operators and the same interpretation of false, a fundamental issue is the notion of *offset* used to adjust the indices of function calls in program lists. It works like 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 $_{-}^{+}$ (**offset_rec(.)(.)** [↗](#)).

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

This function adds n to all the **rec** indices in the expression e . Using the offset operator, the composition of two PVS0 programs (O_1, O_2, \perp, A) and (O_1, O_2, \perp, B) , for short pvs_{0A} and pvs_{0B} , respectively, is expressed by the property:

$$\begin{aligned} \forall (v_i, v_o) : \exists (v) : \gamma(pvs_{0B})(v_i, v) \wedge \gamma(pvs_{0A})(v, v_o) \Leftrightarrow \\ \gamma(O_1, O_2, \perp, [\text{rec}(1, \text{rec}(1 + |A|, \text{vr}))]) :: \text{map}_{-}^{+1}(A) :: \\ \text{map}_{-}^{+(1+|A|)}(B))(v_i, v_o) \end{aligned}$$

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. The lambda notation, used below, is a feature of the PVS specification language.

$$- O_1 := [\lambda(n : \mathbb{N}) : \text{IF } n = 0 \text{ THEN } n \text{ ELSE } n - 1]$$

- $O_2 := [\lambda(n, m : \mathbb{N}) : n \times m]$
- $\perp := 0$

Consider now the PVS0 programs built with these operators and the unitary lists of expressions below specifying respectively the square and the factorial functions, i. e., the 4-tuples $(O_1, O_2, \perp, \text{square})$ and $(O_1, O_2, \perp, \text{factorial})$.

$\text{square} := [\text{op2}(0, \mathbf{vr}, \mathbf{vr})]$
 $\text{factorial} := [\text{ite}(\mathbf{vr}, \text{op2}(0, \mathbf{vr}, \text{rec}(0, \text{op1}(0, \mathbf{vr}))), \text{cnst}(1))]$

The correctness of the composition of *factorial* and *square* using the offset operator, is expressed as the property below.

$$\forall(v_i) : \gamma(O_1, O_2, \perp, \\ [\text{rec}(1, \text{rec}(1 + |\text{factorial}|, \mathbf{vr}))] :: \text{map}_{(-^{+1})}(\text{factorial}) :: \\ \text{map}_{(-^{+(1+|\text{factorial}|)})}(\text{square}))(v_i, (v_i^2)!)]$$

Where, more concretely $\text{map}_{(-^{+1})}(\text{factorial})$ is the expression $\text{ite}(\mathbf{vr}, \text{op2}(0, \mathbf{vr}, \text{rec}(1, \text{op1}(0, \mathbf{vr}))), \text{cnst}(1))$ and $\text{map}_{(-^{+(1+|\text{factorial}|)})}(\text{square})$ is $\text{op2}(0, \mathbf{vr}, \mathbf{vr})$; thus, the list of expressions of the composition is:

$[\text{rec}(1, \text{rec}(2, \mathbf{vr}))] ::$
 $[\text{ite}(\mathbf{vr}, \text{op2}(0, \mathbf{vr}, \text{rec}(1, \text{op1}(0, \mathbf{vr}))), \text{cnst}(1))] ::$
 $[\text{op2}(0, \mathbf{vr}, \mathbf{vr})]$

A functional alternative for the semantic evaluation predicate ε should take into consideration the case in which the evaluation does not return an output. This issue is solved by adding an element to the working type that is interpreted as *none* and denoted by \diamond . The new type, $T \cup \{\diamond\}$, is constructed with the PVS functor **Maybe**, specifically as **Maybe**(T). The evaluation function also includes a parameter that limits the allowed number of nested recursive calls: when this limit is reached, the function returns \diamond . This is given as the function χ (`eval_expr`) in Table 2.

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

$$\forall(pvs_0, e, v_i, v_o) : \varepsilon(pvs_0)(e, v_i, v_o) \Leftrightarrow \\ \exists(n) : \chi(pvs_0)(n, e, v_i) = v_o \wedge v_o \neq \diamond$$

Necessity and sufficiency are formalized as separated lemmas (respectively, `semantic_rel_eval_expr` and `eval_expr_semantic_rel`).

Similarly to [7], a terminating program pvs_0 , satisfies $\forall(v_i) : \exists(v_o) : \gamma(pvs_0)(v_i, v_o)$ or equivalently $\forall(v_i) : \exists(n) : \chi(pvs_0)(n, pvs_0'4(0), v_i) \neq \diamond$.

Two (equivalent) notions of operational semantics for the PVS0 language provide higher flexibility in the formalization since properties may be checked to select one of these notions alternatively. In particular, the semantics provided by the function χ turns clear the measure required in inductive proofs; namely, to show termination of χ , the measure that decreases in each recursive call is the lexicographical order on the pair (n, e) build with the orders on naturals and (sub)expressions. Moreover, this is also the measure used to prove the inductive properties of such a recursive function.

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

Table 2 PVS0 program evaluation function - $pvs_0 = (O_1, O_2, \perp, E_f)$ ([eval_expr](#))
$$\begin{aligned}
\chi(pvs_0)(n, e, v_i) &:= \\
&\text{IF } n = 0 \text{ THEN } \diamond \text{ ELSE CASES } e \text{ OF} \\
&\quad \text{cnst}(v) : v; \\
&\quad \text{vr} : v_i; \\
\text{op1}(j, e_1) &: \text{IF } j < |O_1| \text{ THEN} \\
&\quad \text{LET } v' = \chi(pvs_0)(n, e_1, v_i) \text{ IN} \\
&\quad \text{IF } v' = \diamond \text{ THEN } \diamond \text{ ELSE } O_1(j)(v') \\
&\quad \text{ELSE } \perp; \\
\text{op2}(j, e_1, e_2) &: \text{IF } j < |O_2| \text{ THEN} \\
&\quad \text{LET } v' = \chi(pvs_0)(n, e_1, v_i), \\
&\quad \quad v'' = \chi(pvs_0)(n, e_2, v_i) \text{ IN} \\
&\quad \text{IF } v' = \diamond \vee v'' = \diamond \text{ THEN } \diamond \\
&\quad \text{ELSE } O_2(j)(v', v'') \\
&\quad \text{ELSE } \perp; \\
\text{rec}(j, e_1) &: \text{LET } v' = \chi(pvs_0)(n, e_1, v_i) \text{ IN} \\
&\quad \text{IF } v' = \diamond \text{ THEN } \diamond \\
&\quad \text{ELSIF } j < |E_f| \text{ THEN} \\
&\quad \quad \chi(pvs_0)(n - 1, E_f(j), v') \\
&\quad \text{ELSE } \perp; \\
\text{ite}(e_1, e_2, e_3) &: \text{LET } v' = \chi(pvs_0)(n, e_1, v_i) \text{ IN} \\
&\quad \text{IF } v' = \diamond \text{ THEN } \diamond \\
&\quad \text{ELSIF } v' \neq \perp \text{ THEN} \\
&\quad \quad \chi(pvs_0)(n, e_2, v_i) \\
&\quad \text{ELSE } \chi(pvs_0)(n, e_3, v_i).
\end{aligned}$$

$$\begin{aligned}
\text{valid_index_rec}(e, n) &:= & \text{valid_index}(E_f) &:= \\
\forall(i, e_1) : & & \forall(i < |E_f|) : & \\
\text{subterm}(\text{rec}(i, e_1), e) \Rightarrow i < n & & \text{valid_index_rec}(E_f(i), |E_f|) &
\end{aligned}$$

3 PVS0 Computable and Partial Recursive Classes

We fix the input and output type of PVS0 programs as naturals; thus, O_1 and O_2 , and \perp are unary and binary operators over naturals, and a natural number. Below, a class of partial recursive functions is defined as follows:

$$\begin{aligned}
\text{partial_recursive?}(pvs_0) &:= pvs_0'1 = O_1 \wedge pvs_0'2 = O_2 \wedge \\
&\quad pvs_0'3 = \perp \wedge \text{valid_index}(pvs_0'4)
\end{aligned}$$

In the specifications, O_1 and O_2 are parameters of the theory `mf_pvs0_computable`. Computable *partial_recursive* functions are given as:

$$\text{computable?}(pvs_0) := \text{partial_recursive?}(pvs_0) \wedge T_\varepsilon(pvs_0)$$

Given a *pvs_0* program, if *partial_recursive?*(*pvs_0*) holds, *pvs_0* is of the type `partial_recursive`. If in addition *pvs_0* is terminating, *computable?*(*pvs_0*) holds, and it is of the type `computable`.

The PVS resource that allows building the subtype of a type **T** associated with a predicate, `Pred?` over the type **T** is used to simplify the specification; indeed the subtype is specified as `Pred : TYPE = (Pred?)`. Notice that the types `computable` and `partial_recursive` above are subtypes of the type of PVS0 programs. When

one needs to specify properties and operators over objects of such a subtype, it is only required to refer to such subtypes' parameters (e.g., $\mathbf{x} : (\text{Pred?})$ or $\mathbf{x} : \text{Pred}$). Additionally, PVS allows expanding the properties inherent to a subtype through the proof command `typepred`.

To prove Turing completeness and Rice's Theorem, it is necessary to formalize some lemmas about shift code. As previously, consider PVS0 programs $pvs_{0A} = (O_1, O_2, \perp, A)$ and $pvs_{0B} = (O_1, O_2, \perp, B)$. The first lemma is:

Lemma 1 (Shift code - add_rec_list_aux [↗](#))

$$\begin{aligned} \forall(O_1, O_2, \perp, A, B, e, v_i, n) : \chi(pvs_{0B})(n, e, v_i) = \\ \chi(O_1, O_2, \perp, A :: \text{map}_{(-^{|A|})}(B))(n, e^{|A|}, v_i) \end{aligned}$$

This lemma means that in an evaluation of the expression e considering the PVS0 program pvs_{0B} , it is possible to concatenate a list A in front of B without changing the evaluation semantics, adjusting accordingly the indices in `rec` expressions contained by e and B .

The second lemma is:

Lemma 2 (Shift code - add_rec_list_aux2 [↗](#))

$$\begin{aligned} \forall(O_1, O_2, \perp, B, v_i, n) : \\ \forall(A \mid \text{valid_index}(A)) : \forall(e \mid \text{valid_index_rec}(e, |A|)) : \\ \chi(pvs_{0A})(n, e, v_i) = \chi(O_1, O_2, \perp, A :: B)(n, e, v_i) \end{aligned}$$

This lemma is similar to Lemma 1, but the indices of the `rec` expressions in the evaluated expression e and the list A of the PVS0 program pvs_{0A} must be valid references to a PVS0 expression in A , and the list B of the PVS0 program pvs_{0B} is concatenated in the end.

Both lemmas are proved by induction on the lexicographical order given by pairs (n, e) , built with the orders on naturals and (sub)expressions. The type of pair (n, e) is $\mathbb{N} \times \text{PVS0Expr}$, where `PVS0Expr` is the type of PVS0 expressions. Previous lemmas and the equivalence of ε and χ entails both (respectively, lemmas `add_rec_list` [↗](#) and `add_rec_list2` [↗](#)):

$$\begin{aligned} \forall(O_1, O_2, \perp, A, B, e, v_i, v_o) : \varepsilon(pvs_{0B})(e, v_i, v_o) \Leftrightarrow \\ \varepsilon(O_1, O_2, \perp, A :: \text{map}_{(-^{|A|})}(B))(e^{|A|}, v_i, v_o) \end{aligned}$$

and

$$\begin{aligned} \forall(O_1, O_2, \perp, B, v_i, v_o) : \\ \forall(A \mid \text{valid_index}(A)) : \forall(e \mid \text{valid_index_rec}(e, |A|)) : \\ \varepsilon(pvs_{0A})(e, v_i, v_o) \Leftrightarrow \varepsilon(O_1, O_2, \perp, A :: B)(e, v_i, v_o) \end{aligned}$$

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 a semantic predicate over PVS0 programs is specified as:

$$\begin{aligned} \text{is_semantic_predicate?}(P) := \forall(pvs_{01}, pvs_{02}) : \\ (\forall(v_i, v_o) : \gamma(pvs_{01})(v_i, v_o) \Leftrightarrow \gamma(pvs_{02})(v_i, v_o)) \Rightarrow \\ (P(pvs_{01}) \Leftrightarrow P(pvs_{02})) \end{aligned}$$

If `is_semantic_predicate?` holds for the predicate P , it is said that P is a semantic predicate. For the Gödelization, it is necessary to define a bijective function from each PVS0 expression that works with naturals (thus, the non-interpreted type for PVS0 expressions would be the type of natural numbers) and a bijective function from lists of naturals to naturals. The bijection from expressions to naturals (κ_e) is built over a bijection from pairs of naturals to naturals κ_2 (`tuple2nat` [↗](#)) as below.

$$\kappa_2(m, n) := \frac{(m + n + 1)(m + n)}{2} + n$$

```

κe(len)(e) := CASES e OF
  vr : 0;
  cnst(v) : v × 5 + 1;
  rec(j, e1) : (j + κe(len)(e1) × (len + 1)) × 5 + 2;
  op1(j, e1) : κ2(j, κe(len)(e1)) × 5 + 3;
  op2(j, e1, e2) : κ2(j, κ2(κe(len)(e1), κe(len)(e2))) × 5 + 4;
  ite(e1, e2, e3) : κ2(κe(len)(e1), κ2(κe(len)(e2), κe(len)(e3))) × 5 + 5;

```

In the function κ_e above (`PVS02nat_limit` [↗](#)), for all subexpression `rec(i, e')` of the argument expression e , and i is less or equal than len (the length of the kernel). The reason for this is that the goal is to Gödelize `partial_recursive` programs, and that each index in the `rec` subexpression in an expression in the kernel of the programs must be valid, i. e., be limited by the length of the kernel.

A bijection from lists of naturals to naturals called α was implemented as below.

```

rdc(l) := reverse(cdr(reverse(l)))
αaux(l) := IF |l| = 1 THEN l(0);
           ELSE κ2(αaux(rdc(l)), l(|l| - 1))
α(l) := IF |l| = 0 THEN 0;
         ELSE κ2(|l| - 1, αaux(l)) + 1





```

Above, l is a list of naturals, `reverse` reverses lists and `rdc` deletes the last element of a non-empty list. Notice that α (`listnat2nat` [↗](#)), through applications of α_{aux} (`cons2nat` [↗](#)), transforms recursively the prefix of the input list without the last element into a natural and applies the bijection κ_2 to this natural and the last element of the list. In the Gödelization, the function α_{aux} receives a non-empty list of naturals, each representing an expression in a list of PVS0 expressions. This construction structure becomes similar to the ones previously used and ease the proof of the Recursion Theorem. In particular, it will be helpful when α is used in inductive proofs in which PVS0 programs are built, adding to the kernel a constant that represents a number associated with the Gödelization of a list of expressions.

Using α , a bijection, from the class of partial recursive functions to naturals, is given as below.

$$\kappa_p(pvs_0) := \alpha(\text{map}(\kappa_e(|pvs_0'|4| - 1))(pvs_0'4)) - 1$$

The function κ_p (`p_recursive2nat` [↗](#)) Gödelizes the `partial_recursive` PVS0 programs.

To prove bijectivity of κ_p , it was necessary to build the inverses of κ_2 , κ_e , α_{aux} (and α) given respectively as κ_2^{-1} ([nat2tuple](#) ) , κ_e^{-1} , α_{aux}^{-1} and α^{-1} (respectively, [nat2PVS0_limit](#) , [nat2listnat_aux](#)  and [nat2listnat](#) ). But bijectivity is only required for the Fixed-Point Theorem. The formalizations of Rice's and Recursion theorems as well as undecidability of the Halting Problem use also κ_p , but they do not use its bijectivity; any Gödelization function can be used. These inverses were specified as below.


$$\begin{aligned} \kappa_2^{-1}(i) := & \text{IF } i = 0 \text{ THEN } (0, 0) \\ & \text{ELSIF } \kappa_2^{-1}(i - 1)'1 = 0 \text{ THEN } (\kappa_2^{-1}(i - 1)'2 + 1, 0); \\ & \text{ELSE } (\kappa_2^{-1}(i - 1)'1 - 1, \kappa_2^{-1}(i - 1)'2 + 1) \end{aligned}$$

$$\begin{aligned} \alpha_{aux}^{-1}(len, n) := & \text{IF } len = 0 \text{ THEN } [n]; \\ & \text{ELSE } \alpha_{aux}^{-1}(len - 1, \kappa_2^{-1}(n)'1) :: [\kappa_2^{-1}(n)'2] \end{aligned}$$

$$\begin{aligned} \alpha^{-1}(n) := & \text{IF } n = 0 \text{ THEN } []; \\ & \text{ELSE } \alpha_{aux}^{-1}(\kappa_2^{-1}(n - 1)'1, \kappa_2^{-1}(n - 1)'2) \end{aligned}$$

$$\begin{aligned} \kappa_e^{-1}(len)(n) := & \\ & \text{IF } n = 0 \quad \text{THEN vr} \\ & \text{ELSIF } (n - 1)|5 \text{ THEN } \text{cnst}(\frac{n - 1}{5}) \\ & \text{ELSIF } (n - 2)|5 \text{ THEN } \text{rec}(\frac{n - 2}{5} \% (len + 1), \kappa_e^{-1}(len)(\lfloor \frac{n - 2}{5 \times (len + 1)} \rfloor)) \\ & \text{ELSIF } (n - 3)|5 \text{ THEN } \text{op1}(\kappa_2^{-1}(\frac{n - 3}{5})'1, \kappa_e^{-1}(len)(\kappa_2^{-1}(\frac{n - 3}{5})'2)) \\ & \text{ELSIF } (n - 4)|5 \text{ THEN } \text{op2}(\kappa_2^{-1}(\frac{n - 4}{5})'1, \\ & \quad \kappa_e^{-1}(len)(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n - 4}{5})'2))'1, \\ & \quad \kappa_e^{-1}(len)(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n - 4}{5})'2)'2)) \\ & \text{ELSE} \quad \text{ite}(\kappa_e^{-1}(len)(\kappa_2^{-1}(\frac{n - 5}{5})'1), \\ & \quad \kappa_e^{-1}(len)(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n - 5}{5})'2))'1, \\ & \quad \kappa_e^{-1}(len)(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n - 5}{5})'2)'2)) \end{aligned}$$

In the function α_{aux}^{-1} , len is a natural that defines the length ($len - 1$) of the list of naturals. In the function κ_e^{-1} , the argument len is the length of the indices of the **rec** expressions, $a \% b$ denotes the remainder of a divided by b , $a|b$ the predicate a divides b and $\lfloor a \rfloor$ the floor of a .

The function κ_e^{-1} ([nat2PVS0_limit](#) ) uses subtype predicates that is an interesting feature of PVS used in the specification of recursive function. The type of the image of the function κ_e^{-1} is specified as the type of **PVS0** expressions whose recursive subexpressions have indices less than or equal to len . PVS generates a Type Correctness Condition (TCC) that is a proof obligation stating that this is indeed the type of outputs computed by the specified function. Having this property as a proved TCC simplifies further formalizations of properties of κ_e^{-1}

because typing conditions of the outputs of κ_e^{-1} guarantee Gödelizations of PVS0 expressions that have recursive calls with valid indices.

The inverse of κ_p is specified as below.

$$\kappa_p^{-1}(n) := (O_1, O_2, \perp, \text{map}(\kappa_e^{-1}(|\alpha^{-1}(n+1)| - 1))(\alpha^{-1}(n+1)))$$

These functions are formalized to be right/left inverses according to Lemma 3.

Lemma 3 (κ_e and α_{aux} Inversibility - PVS02nat_nat2PVS0_limit [↗](#), nat2PVS0_PVS02nat_limit [↗](#), nat2listnat_aux_cons2nat [↗](#) and cons2nat_nat2listnat_aux [↗](#))

Left and right inversibility of the operators κ_e and α_{aux} is formalized as:

1. $\forall n, len : \kappa_e(len)(\kappa_e^{-1}(len)(n)) = n$ and $\forall e, len : \kappa_e^{-1}(len)(\kappa_e(len)(e)) = e$
2. $\forall l : \alpha_{aux}^{-1}(|l| - 1, \alpha_{aux}(l)) = l$ and $\forall len, n : \alpha_{aux}(\alpha_{aux}^{-1}(len, n)) = n$

4 Turing Completeness of the Model

In order to achieve a Turing complete model, a class of partial recursive functions is specified in which the built-in operators include the functions successor, projections, greater-than and the function κ_2 . Projections are built using the inverse of the function κ_2 .

The successor and greater-than functions, as well as projections of the elements of the tuple given by a natural, are given below.

$$\begin{aligned} \text{succ}(n) &:= n + 1 \\ \text{greater}(m, n) &:= \text{IF } m > n \text{ THEN } 1 \text{ ELSE } 0 \\ \pi_1(n) &:= ((\lambda(m, n : \mathbb{N}) : m) \circ \kappa_2^{-1})(n) \\ \pi_2(n) &:= ((\lambda(m, n : \mathbb{N}) : n) \circ \kappa_2^{-1})(n) \end{aligned}$$

The fixed built in operators are: $O_1 := [\text{succ}, \pi_1, \pi_2]$ and $O_2 := [\text{greater}, \kappa_2]$, and \perp is interpreted as $0 \in \mathbb{N}$.

For brevity we will use the predicate below (that does not belong to the specification) for the class of PVS0 programs having O_1, O_2 and 0 as parameters.

$$\begin{aligned} \text{partial_recursive}_{TC}(pvs_0) &:= pvs_0'1 = O_1 \wedge pvs_0'2 = O_2 \wedge \\ & pvs_0'3 = 0 \wedge \text{valid_index}(pvs_0'4) \end{aligned}$$

Also, for brevity, any PVS0 program pvs_0 that belongs to the above predicate is said to be of type $\text{partial_recursive}_{TC}$. This type is obtained as a parameterization of the type partial_recursive . In the specification, to define the type $\text{partial_recursive}_{TC}$ it is enough to pass the above parameters to the theory $\text{mf_pvs0_computable}$. [↗](#) In order to show Turing completeness of the class of PVS0 programs of such type, it is only necessary to prove that there are implementations of the constant, successor, and projection functions and that the class is closed under composition, minimization and primitive recurrence. The most difficult cases in this formalization are those related to the implementation of projection and to the class's closedness under composition, minimization, and primitive recurrence.

The n -tuples in PVS are specified as lists of naturals, but encoded as a unique natural.

The function *nat2list* below, transforms uniquely a natural m into a list of naturals of length n .

$$\begin{aligned} \text{nat2list}(n, m) &:= \\ &\text{IF } n = 0 \text{ THEN } []; \\ &\text{ELSIF } n = 1 \text{ THEN } [m]; \\ &\text{ELSE } [\kappa_2^{-1}(m)'1] :: \text{nat2list}(n - 1, \kappa_2^{-1}(m)'2); \end{aligned}$$

The following abbreviations are used:

$$\begin{aligned} \text{succ}^S(e) &:= \text{op1}(0, e); \quad \pi_1^S(e) := \text{op1}(1, e); \quad \pi_2^S(e) := \text{op1}(2, e); \\ \text{greater}^S(e_1, e_2) &:= \text{op2}(0, e_1, e_2); \\ \kappa_2^S(e_1, e_2) &:= \text{op2}(1, e_1, e_2). \end{aligned}$$

Since built-in operators and zero (to interpret \perp) are fixed for any PVS0 program *pvs0* of type `partial_recursiveTC`, the focus would be on the program itself, i.e., on *pvs0*'4.

The PVS0 `partial_recursiveTC` program *equal*, specified below, verifies if the pair of naturals encoded as a unique natural are equal.

$$\begin{aligned} &\text{LET } i = \pi_1^S(\mathbf{vr}), \quad j = \pi_2^S(\mathbf{vr}) \text{ IN} \\ &\text{equal}'4 := \\ &[\text{ite}(\text{greater}^S(i, j), \\ &\quad \text{cnst}(0), \\ &\quad \text{ite}(\text{greater}^S(j, i), \text{cnst}(0), \text{cnst}(1)))] \end{aligned}$$

Some technical PVS0 `partial_recursiveTC` programs were specified to deal with projections of naturals' tuples encoded as naturals.

The PVS0 `partial_recursiveTC` program *proj_aux*, specified below, receives as input a natural that codifies a quadruple of naturals (i, j, k, l) , and outputs the $(j - i)$ -t projection of l . Case $k = j$ the input is seen as a $(j - i)$ -tuple, otherwise, it is seen as a tuple of length greater than $(j - i)$. In this specification, the function k_4 is used to encode a quadruple of naturals as a natural used in the recursive calls, allowing in this manner the increment of the first element of the quadruple $(i, \text{succ}^S(i), \dots)$ and advancing by the second projection of the fourth element $(l, \pi_2^S(l), \dots)$.

$$\begin{aligned} &\text{LET } i = \pi_1^S(\mathbf{vr}), \quad j = \pi_1^S(\pi_2^S(\mathbf{vr})), \quad k = \pi_1^S(\pi_2^S(\pi_2^S(\mathbf{vr}))), \\ &\quad l = \pi_2^S(\pi_2^S(\pi_2^S(\mathbf{vr}))), \quad k_4(x, y, z, w) = \kappa_2^S(x, \kappa_2^S(y, \kappa_2^S(z, w))) \text{ IN} \\ &\text{proj_aux}'4 := \\ &[\text{ite}(\text{greater}(j, i), \\ &\quad \text{rec}(0, k_4(\text{succ}^S(i), j, k, \pi_2^S(l))), \\ &\quad \text{ite}(\text{rec}(1, \kappa_2^S(j, k)), l, \pi_1^S(l)))] :: \text{map}(-^+)(\text{equal}'4) \end{aligned}$$

The PVS0 `partial_recursiveTC` program *proj* uses *proj_aux* to receive as input a natural that codifies a triple of naturals (i, j, k) , and outputs the i -th projection of k (seen k as a $j + 1$ tuple).

$$\begin{aligned} &\text{LET } i = \pi_1^S(\mathbf{vr}), \quad j = \pi_1^S(\pi_2^S(\mathbf{vr})), \quad k = \pi_2^S(\pi_2^S(\mathbf{vr})), \\ &\quad k_4(x, y, z, w) = \kappa_2^S(x, \kappa_2^S(y, \kappa_2^S(z, w))) \text{ IN} \\ &\text{proj}'4 := \\ &[\text{rec}(1, k_4(\text{cnst}(0), i, j, k))] :: \text{map}(-^+)(\text{proj_aux}'4) \end{aligned}$$

The correctness of the PVS0 `partial_recursiveTC` program for projection, `proj`, is formalized as Lemma 4. The lemma shows that `proj` projects correctly the i -th element of any tuple encoded as the natural m (seen as an $n + 1$ tuple).

Lemma 4 (Correctness of Projection - `proj_correctness` [↗](#))

$$\forall(i, m) : \forall(n \mid i \leq n) : \gamma(\text{proj})(\kappa_2(i, \kappa_2(n, m)), \text{nat2list}(n + 1, m)(i))$$

The analysis of composition requires the functions `exprComp` and `chainOffset` below. In these functions, l is a non-empty list of say m list of expressions that are the kernel of PVS0 programs. The idea is to simulate the composition of an m -ary function with m functions. As can be observed in Section 2, the composition of two PVS0 programs of the same class of partial recursive functions is straightforward. Nevertheless, to show Turing completeness, the composition must be specified between a PVS0 program and an m -tuple of PVS0 programs. To specify an n -tuple of an arbitrary length, non-empty lists are used.

$$\begin{aligned} \text{exprComp}(n, l) &:= \\ &\text{IF } |l| = 1 \text{ THEN } \text{rec}(n, \text{vr}); \\ &\text{ELSE } \kappa_2^S(\text{rec}(n, \text{vr}), \text{exprComp}(n + |l(0)|, \text{cdr}(l))) \end{aligned}$$

$$\begin{aligned} \text{chainOffset}(n, l) &:= \\ &\text{IF } |l| = 1 \text{ THEN } \text{map}(-^{+n})(l(0)); \\ &\text{ELSE } \text{map}(-^{+n})(l(0)) :: \text{chainOffset}(n + |l(0)|, \text{cdr}(l)); \end{aligned}$$

Let F be a PVS0 program, L a non-empty list of PVS0 programs, and $l := \text{map}(\lambda(x, y, z, w) : w)(L)$. To specify composition, a new list of PVS0 expressions is created, where the head of this new list is a recursive expression that calls the expression f , and the expressions in the tail are given by l . In addition, the function `chainOffset` (`chain_offset` [↗](#)) adjusts the indices of the PVS0 expressions of F and L in the composition. When evaluating this new list of expressions, i.e., the new PVS0 program, the function `exprComp` (`expr_comp` [↗](#)) generates a PVS0 expression whose evaluation codifies a list of naturals (which are the results of the application of the PVS0 programs in L to the input) into a natural. This natural is then passed as an input parameter to evaluate F .

$$\begin{aligned} \text{comp}(F'4, l)'4 &:= [\text{rec}(1, \kappa_2^S(\text{cnst}(|l|), \\ &\quad \text{exprComp}(1 + |F'4|, l))) :: \\ &\quad \text{chainOffset}(1, [F'4] :: l)] \end{aligned}$$

Finally, the composition lemma also requires a way to represent n -tuples of naturals (formalized as non empty list of naturals) into naturals:

$$\begin{aligned} \text{list2nat}(l_n) &:= \\ &\text{IF } |l_n| = 1 \text{ THEN } l_n(0); \\ &\text{ELSE } \kappa_2(l_n(0), \text{list2nat}(\text{cdr}(l_n))); \end{aligned}$$

Now, it is possible to establish the correctness of composition lemma for a PVS0 kernel f and a list of kernels l as follows.

Lemma 5 (Correctness of Composition - comp_is_composition [↗](#))

$$\begin{aligned}
& \forall(f, l \mid |l| > 0) : \forall(v_i, v_o) : \\
& \quad \gamma(\mathit{comp}(f, l))(v_i, v_o) \Leftrightarrow \\
& \quad \exists(l_n \mid |l_n| = |l|) : \\
& \quad \quad \forall(i \mid i < |l_n|) : \gamma(\mathbf{O}_1, \mathbf{O}_2, 0, l(i))(v_i, l_n(i)) \wedge \\
& \quad \quad \gamma(\mathbf{O}_1, \mathbf{O}_2, 0, f)(\kappa_2(|l_n|, \mathit{list2nat}(l_n)), v_o)
\end{aligned}$$

The formalization of correctness of composition (Lemma 5) is by induction on the length of l . The proof requires some technical lemmas, such as showing that the indices of function calls used by `rec`, generated by the functions `exprComp` and `chainOffset`, are valid. This guarantees that `comp` generates a PVS0 `partial_recursiveTC` program indeed.

The lemma on correctness of minimization of `partial_recursiveTC` PVS0 programs uses the function `min_aux` specified below. This function receives as input the list of expressions of a PVS0 program f , and gives as output a PVS0 program that for a given natural that encodes a pair of naturals (i, j) outputs a natural k such that $i \leq k$, and f applied to $\kappa_2(k, j)$ computes zero, and for all naturals such that $i \leq m < k$ f applied to $\kappa_2(m, j)$ is defined and greater than zero. It is necessary to pass as parameter a natural encoding a pair (m, j) because the minimization deals with n -ary functions, being one of the arguments m , and the remaining $n - 1$ arguments encoded by j .

$$\begin{aligned}
\mathit{min_aux}(f'4)'4 := & \\
& [\mathit{ite}(\mathit{rec}(1, \mathbf{vr}), \\
& \quad \mathit{rec}(0, \kappa_2^S(\mathit{succ}^S(\pi_1^S(\mathbf{vr})), \pi_2^S(\mathbf{vr}))), \\
& \quad \pi_1^S(\mathbf{vr}))] :: \\
& \mathit{map}(_{+1})(f'4)
\end{aligned}$$

Using `min_aux`, the (list of expressions of the) minimization of f is specified below.

$$\mathit{min}(f'4)'4 := [\mathit{rec}(1, \kappa_2^S(\mathit{cnst}(0), \mathbf{vr}))] :: \mathit{map}(_{+1})(\mathit{min_aux}(f'4)'4)$$

The following lemma states that `min` is indeed the desired minimization.

Lemma 6 (Correctness of Minimization - min_correctness [↗](#))

$$\begin{aligned}
& \forall(f, j, k) : \\
& \quad \gamma(\mathit{min}(f'4))(j, k) \Leftrightarrow \\
& \quad (\gamma(f)(\kappa_2(k, j), 0) \wedge \\
& \quad \forall(m \mid m < k) : \exists(v_o \mid v_o > 0) : \gamma(f)(\kappa_2(m, j), v_o))
\end{aligned}$$

To show correctness of primitive recurrence, cut-off subtraction of a pair of naturals encoded as a unique natural is specified as:

$$\begin{aligned}
& \mathbf{LET} \ i = \pi_1^S(\mathbf{vr}), \ j = \pi_2^S(\mathbf{vr}) \ \mathbf{IN} \\
& \mathit{sub}'4 := \\
& \quad [\mathit{ite}(\mathit{greater}^S(i, j), \\
& \quad \quad \mathit{succ}^S(\mathit{rec}(0, \kappa_2^S(i, \mathit{succ}^S(j)))), \\
& \quad \quad \mathit{cnst}(0))]
\end{aligned}$$

Using *sub*, the cut-off subtraction by 1 is specified:

$$\begin{aligned} \text{sub1}'4 &:= \\ &[\mathbf{rec}(1, \kappa_2^S(\mathbf{vr}, \mathbf{cnst}(1)))] :: \text{map}(-+1)(\text{sub}'4) \end{aligned}$$

The primitive recurrence, *prim_recur*, is given by the PVS0 program:

```

LET  i = π1S(vr),
     j = π2S(vr),
     less1(x) = rec(1 + |recur'4| + |final'4|, x),
     recur_fun(x, y, z) = rec(1, κ2S(x, κ2S(y, z))),
     final_fun(x) = rec(1 + |recur'4|, x),
     recur_call(x, y) = rec(0, κ2S(x, y))
IN
prim_recur(recur'4, final'4)'4 :=
[ite(i,
  recur_fun(recur_call(less1(i), j), less1(i), j),
  final_fun(j))] ::
map(-+1)(recur'4) :: map(-+1 + |recur'4|)(final) ::
map(1 + |recur'4| + |final'4|)(sub1'4)

```

The function *prim_recur* receives two kernels of `partial_recursiveTC` programs, *recur'4* and *final'4*, and returns another `partial_recursiveTC` program that implements primitive recurrence for functions *r* and *f* as below.

$$\begin{aligned} \rho(r, f)(0, j_1, \dots, j_m) &:= f(j_1, \dots, j_m) \\ \rho(r, f)(i + 1, j_1, \dots, j_m) &:= r(\rho(r, f)(i, j_1, \dots, j_m), i, j_1, \dots, j_m) \end{aligned}$$

The lemma below, states that *prim_recur* is indeed primitive recurrence.

Lemma 7 (Primitive Recurrence Correctness - `prim_recur_correctness`)



$$\begin{aligned} \forall(\text{recur}, \text{final}) : \forall(i, j, \text{ans}) : \\ \gamma(\text{prim_recur}(\text{recur}'4, \text{final}'4))(\kappa_2(i, j), \text{ans}) \Leftrightarrow \\ \exists(l_n \mid i + 1 = |l_n|) : \text{ans} = l_n(|l_n| - 1) \wedge \\ \gamma(\text{final})(j, l_n(0)) \wedge \\ \forall(k \mid k < |l_n| - 1) : \\ \gamma(\text{recur})(\kappa_2(l_n(k), \kappa_2(k, j)), l_n(k + 1)) \end{aligned}$$

As correctness of composition (Lemma 5), the formalization of correctness of minimization and primitive recursion (Lemmas 6 and 7) require additional technical elements such as the inductive predicates below that avoid expansions of the predicate γ resulting in expansions of the evaluation predicate ε .

```

min_relation(i, j, f, ans) :=
  IF γ(f)(κ2(i, j), 0) THEN ans = i;
  ELSIF ∃(k) : γ(f)(κ2(i, j), k)
  THEN min_relation(i + 1, j, f, ans);
  ELSE False.

```

$$\begin{aligned}
& \text{prim_recur_relation}(\text{recur}, \text{final})(i, j)(\text{ans}) := \\
& \text{IF } i \neq 0 \text{ THEN } \exists(z) : \\
& \quad \gamma(\text{recur})(\kappa_2(z, \kappa_2(i-1, j)), \text{ans}) \wedge \\
& \quad \text{prim_recur_relation}(\text{recur}, \text{final})(i-1, j)(z); \\
& \text{ELSE } \gamma(\text{final})(j, \text{ans}).
\end{aligned}$$

Using these predicates, one avoids exhaustive expansions of the ε predicate and thus the generation of existential goals that would require concrete instantiations. In contrast, using the inductive predicates *min_relation* and *prim_recur_relation*, above, PVS will generate inductive schemes, in which no expansion of γ would be required, simplifying in this manner the formalization.

The sufficiency of the correctness of minimization is formalized using the inductive schema given by the predicate *min_relation*. The necessity is formalized using the equivalence between the evaluation function χ and the predicate ε . As discussed in the end of Section 2, the function χ provides the measure to be applied in inductive proofs like the one performed for formalizing the necessity. Similarly, the sufficiency of the correctness of the primitive recurrence is formalized using the predicate *prim_recur_relation*, while a straightforward induction on i proves necessity.

5 The Recursion Theorem

The Recursion Theorem states that for any PVS0 list of expressions E_f there exists a partial recursive PVS0 program such that they both can be used to build another partial recursive program that outputs its Gödel number. This means that there are PVS0 programs that can calculate their own Gödel numbers and process them according to implementations provided by the programmer. Notice that the Recursion Theorem holds for any list of expressions E_f without requiring that *valid_index*(E_f) holds. In Turing complete models, it is possible to design entities that print themselves. From this property, depending on the chosen lists of unary and binary operators, if it is possible to create a partial recursive PVS0 program from a list of PVS0 expressions such that its output for any evaluation is itself, then the Rice's Theorem holds.

The formalization uses the same basic operators for the successor, projection, greater-than, and the bijection κ_2 operators applied to formalize Turing Completeness for the PVS0 model. We dispose of constructions obtained in the proof of Turing Completeness such as composition, minimization and primitive recurrence. However, in the formalization, we opt for building the required constructions implementing PVS0 programs directly. These programs are ensembled using simultaneously several PVS0 programs simplifying in this manner the constructions. The result is specified as below.

Theorem 1 (Recursion Theorem - Recursion_Theorem [↗](#))

$$\begin{aligned}
& \forall(E_f) : \exists(\text{print} : \text{partial_recursive}) : \\
& \text{LET } \text{self} = (\mathbf{O}_1, \mathbf{O}_2, 0, E_f :: \text{map}_{-}^{+|E_f|}(\text{print}'4)) \text{ IN} \\
& \quad \text{partial_recursive}_{TC}(\text{self}) \wedge \\
& \quad \forall(i) : \varepsilon(\text{self})(\text{print}'4(0)^{+|E_f|}, i, \kappa_p(\text{self}))
\end{aligned}$$

Proof To build *print*, the same idea of programming computing viruses is followed. A list of expressions to calculate the Gödel number of *self* is added to E_f . In this manner one guarantees the desired behavior of *self* that is to be able to calculate its own Gödel number, as a quine does, but also to process it accordingly to the programmers desire. Thus, the kernel of *self* can be split in three parts: E_f , a second part A , and $[\mathbf{cnst}(\alpha_{aux}(\mathit{map}(\kappa_e(|E_f :: A|))(E_f :: A)))]$, such that $\mathit{self}'4 = E_f :: A :: [\mathbf{cnst}(\alpha_{aux}(\mathit{map}(\kappa_e(|E_f :: A|))(E_f :: A)))]$. The last expression in the kernel of *self* contains a constant number associated to the Gödel number of $E_f :: A$. The part A calls this last expression and uses this result to calculate the Gödel number of *self*. Finally, the part E_f uses the Gödel number of *self* accordingly to the programmer desire.

The function α_{aux} was recursively specified from the back to the front to be adapted to $\mathit{self}'4$ (in which a natural, related to the first element, is calculated before another natural, related to the last element, is calculated). This decision reduces the effort necessary in the formalization since it allows avoiding the elaborated analysis that a recursive specification from the front to the back would imply. In such an alternative version of $\mathit{self}'4$ the last element should represent a stack of naturals associated to each element in $E_f :: A$ by the function κ_e . In such a case, to calculate the Gödel number of the alternative version of *self* it would be necessary to add the number associated with its last element to the bottom of the stack.

The second part of *self*, A , is defined as below, where δ is the greatest index of **rec** found in the list E_f , using the function *printA*:

$$A := \mathit{printA}(\delta, |E_f|)^{+|E_f|}$$

The function *printA* is specified below.

$$\begin{aligned} \mathit{printA}(\mathit{len}, \mathit{len2}) := & \\ & [\kappa_2^S(\mathbf{cnst}(1 + \mathit{len} + \mathit{len2} + |\mathit{mult}|), \kappa_2^S(\mathbf{rec}(|\mathit{mult}| + \mathit{len} + 1, \mathbf{vr}), \\ & \mathit{succ}^S(\mathbf{rec}(1, \kappa_2^S(\mathbf{cnst}(5), \mathbf{rec}(|\mathit{mult}| + \mathit{len} + 1, \mathbf{vr}))))))] :: \\ & \mathit{mult}^{+1} :: [\mathbf{vr}]^{\mathit{len}} \end{aligned}$$

Above $[\mathbf{vr}]^{\mathit{len}}$ is a list with len repetitions of \mathbf{vr} . The list for mult is specified to receive a natural number as input, apply the bijective function κ_2^{-1} to obtain a pair of naturals and multiplying them, as below.

$$\begin{aligned} \mathit{mult} := & \\ & [\mathbf{ite}(\pi_1^S(\mathbf{vr}), \\ & \quad \mathbf{rec}(1, \kappa_2^S(\pi_2^S(\mathbf{vr}), \mathbf{rec}(0, \kappa_2^S(\mathbf{rec}(1 + |\mathit{sum}|, \pi_1^S(\mathbf{vr})), \pi_2^S(\mathbf{vr}))))), \\ & \quad \mathbf{cnst}(0))] \\ & :: \mathit{sum}^{+1} :: \mathit{sub1}'4^{+1+|\mathit{sum}|} \end{aligned}$$

Since in the specification of A above the arguments of *printA* are δ and $|E_f|$, it is warranted that the indices of **rec** in *self* are always valid. Thus, *self* is **partial_recursive**_{TC} because the restriction on basic operator is maintained by construction.

The list mult , used in the specification of *printA*, multiplies using sum that adds pairs of naturals encoded as a unique natural by the function κ_2 as below.

$$\begin{aligned} sum &:= [\text{ite}(\pi_1^S(\mathbf{vr}), \\ &\quad \text{succ}(\text{rec}(0, \kappa_2^S(\text{rec}(1, \pi_1^S(\mathbf{vr})), \pi_2^S(\mathbf{vr}))), \\ &\quad \pi_2^S(\mathbf{vr}))], \\ &:: \text{sub1}'4^{+1} \end{aligned}$$

Although *sum* and *mult* are simple, their codifications as PVS0 programs require also verifying their correctness. This is achieved proving that these functions are functionally equivalent to the PVS functions specified as below.

$$\begin{aligned} sum_f(x, y) &= \text{IF } x \neq 0 \text{ THEN } 1 + sum_f(x - 1, y) \text{ ELSE } y \\ mult_f(x, y) &= \text{IF } x \neq 0 \text{ THEN } y + mult_f(x - 1, y) \text{ ELSE } 0 \end{aligned}$$

Formalizing correctness of *mult* and *sum* directly is possible, but hard-to-follow because the semantic evaluation generates a large chain of existential quantifiers. Thus, the equivalence between the PVS0 specified code and their associated PVS functions was formalized equivalent as a simple alternative. Also, the correctness of the associated PVS functions was showed. Thus, the correctness of *mult* and *sum* are given as corollaries.

The correctness of *printA* is given as next lemma.

Lemma 8 (Correctness of *printA* - `print_correctness` [↗](#))

$$\begin{aligned} \forall(i, len, len2, h) : \\ \gamma(\mathbf{O}_1, \mathbf{O}_2, 0, \text{printA}(len, len2) :: [\text{cnst}(h)]) \\ (i, \kappa_2(1 + len + len2 + |mult|, \kappa_2(h, 5 \times h + 1))) \end{aligned}$$

To use this lemma, *len*, *len2* and *h* are instantiated respectively as δ , $|E_f|$ and $\alpha_{aux}(\text{map}(\kappa_e(|E_f| :: A))(E_f :: A))$, where $\text{self}'4 := E_f :: A :: [\text{cnst}(h)]$. This gives:

$$\begin{aligned} \forall(i) : \\ \gamma(\mathbf{O}_1, \mathbf{O}_2, 0, \text{printA}(\delta, |E_f|) :: [\text{cnst}(h)]) \\ (i, \kappa_2(|\text{self}'4| - 1, \kappa_2(h, \kappa_e(|\text{self}'4| - 1)(\text{cnst}(h))))) \end{aligned}$$

because,

$$\begin{aligned} 1 + \delta + |E_f| + |mult| &= |\text{self}'4| - 1 \\ 5 \times h + 1 &= \kappa_e(|\text{self}'4| - 1)(\text{cnst}(h)) \end{aligned}$$

The expression $\kappa_2(h, \kappa_e(|\text{self}'4| - 1)(\text{cnst}(h)))$ can be replaced by $\alpha_{aux}(\text{map}(\kappa_e(|\text{self}'4| - 1))(\text{self}'4))$ because expanding the definition of *map*, α_{aux} , and *self*, one has:

$$\begin{aligned} \alpha_{aux}(\text{map}(\kappa_e(|\text{self}'4| - 1))(\text{self}'4)) &= \\ \alpha_{aux}(\text{map}(\kappa_e(|\text{self}'4| - 1))(E_f :: A :: [\text{cnst}(h)])) &= \\ \alpha_{aux}(\text{map}(\kappa_e(|\text{self}'4| - 1))(E_f :: A) :: \kappa_e(|\text{self}'4| - 1)(\text{cnst}(h))) &= \\ \kappa_2(\alpha_{aux}(\text{map}(\kappa_e(|\text{self}'4| - 1))(E_f :: A)), \kappa_e(|\text{self}'4| - 1)(\text{cnst}(h))) &= \\ \kappa_2(h, \kappa_e(|\text{self}'4| - 1)(\text{cnst}(h))) \end{aligned}$$

The result of this replacement is:

$$\begin{aligned} \forall(i) : \\ \gamma(\mathbf{O}_1, \mathbf{O}_2, 0, \text{printA}(\delta, |E_f|) :: [\text{cnst}(h)]) \\ (i, \kappa_2(|\text{self}'4| - 1, \alpha_{aux}(\text{map}(\kappa_e(|\text{self}'4| - 1))(\text{self}'4)))) \end{aligned}$$

Then, $\alpha_{aux}(\text{map}(\kappa_e(|self'4| - 1))(self'4))$ can be replaced by $\kappa_p(self)$ because by definition of κ_p and α the equalities below hold.

$$\begin{aligned} \kappa_p(self) &= \\ \alpha(\text{map}(\kappa_e(|self'4| - 1))(self'4)) - 1 &= \\ \kappa_2(|self'4| - 1, \alpha_{aux}(\text{map}(\kappa_e(|self'4| - 1))(self'4))) & \end{aligned}$$

Thus, it can be concluded that:

$$\begin{aligned} \forall(i) : \\ \gamma(\mathbf{O}_1, \mathbf{O}_1, 0, \text{printA}(\delta, |E_f|) :: [\mathbf{cnst}(h)])(i, \kappa_p(self)) \end{aligned}$$

And finally, by application of the shift code lemmas (Lemmas 1 and 2), expanding γ and adding E_f in front of $\text{printA}(\delta, |E_f|) :: [\mathbf{cnst}(h)]$, one concludes the proof of the theorem. \square

As discussed before Theorem 1, instead of using composition, minimization and primitive recurrence operators implemented for the proof of Turing Completeness in Section 4, the formalization approach is based on the direct implementation of PVS0 programs. This decision allows the analysis of computational properties directly over the PVS0 model, avoiding using the theory of partial recursive functions. Of course, the composition, minimization and primitive recurrence operators may be applied for constructing PVS0 programs such as *sum*, *mult* and others (used in the formalization of Recursion Theorem in Section 5). Notice that this kind of construction turns difficult to understand the semantics of the programs. For instance, an alternative implementation of the program *sum* given in Section 5, can be done using composition and primitive recurrence as below.

$$\begin{aligned} \text{sum} := \text{prim_recur}(\text{comp}([\text{succ}(\mathbf{vr})], [[\text{comp}(\text{proj}'4, \kappa_2^S(0, \kappa_2^S(2, \mathbf{vr}))]'4]]'4), \\ \text{comp}(\text{proj}'4, [[\kappa_2^S(0, \kappa_2^S(0, \mathbf{vr}))]]'4)) \end{aligned}$$

The construction of the specialized Gödelization functions required to build *self* is the most difficult part of this formalization. One challenge in the implementation of κ_p was to build it in such a manner that it facilitates further steps of the formalization. An appropriate function α_{aux} was enough to reach this aim. Specifically for the Recursion Theorem, it is not required κ_p to be bijective. However, it was done in this way (in the file `mf_pvs0_Computable` of the theory, see Figure 1) in order to make it useful for the formalization of other theorems such as the Fixed-Point Theorem. Ensuring that κ_p is bijective was technically difficult since it requires that all necessary auxiliary functions were also bijective. For some auxiliary components, the formalization was straightforward. However, for other ones, PVS infers some types such that the application of some lemmas about lists (of PVS0 expressions, i.e., kernel of PVS0 programs, and naturals) do not work. Such types came to arise when specific kernels of PVS0 programs were considered, such as those that included only valid recursive call indices. An example of such properties on lists is $|A :: B| = |A| + |B|$. There is an appropriate lemma for this property, but considering the types of A, B and $A :: B$ the same. Nevertheless, if the types of A and B are a subtype of $A :: B$ bein all them inputs of the function $\text{length}(|_| : [T \rightarrow \text{nat}])$, the lemma needs to be specialized and proved separately. The general, non-provided in PVS, solution is to prove that if S is a subtype of


T , and $A : S$, then $|A|[T] = |A|[S]$. Several similar type inference problems were solved when formalizing the Recursion Theorem (in theories that for simplicity are not included in Figure 1).

6 Rice's Theorem

The formalization of Rice's Theorem is a corollary of the Recursion Theorem. It is proved that if using the basic built-in operators, the Recursion Theorem holds then Rice's Theorem for this theory also holds. Notice, that the basic operators used in theory `mf_pvs0.Recursion.Theorem` to guarantee this theorem, and those used in theory `mf_pvs0.Turing.Completeness` to ensure Turing Completeness are the same. The formalization in this section proves that for all Gödelizations that the Recursion Theorem holds then also Rice's Theorem holds. Similarly to standard demonstrations of the undecidability of the Halting Problem and the uncountability of real numbers, the formalization is based on a Cantor's diagonal argument. An alternative formalization approach is based on the construction of a universal program for the PVS0 model. Still, it would increase the complexity of the formalization. Using such proof strategy, the formalization would require the construction of an elaborated reduction of the Halting Problem to the problem of separability of semantic properties of PVS0 programs. Thus, the current formalization does not use (undecidability of) the Halting Problem and depends only on the above-mentioned Theorem 1.

6.1 Formalization of Rice's Theorem

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

Theorem 2 (Rice's Theorem - `Rice.theorem_for_Turing_complete_pvs0`) 

$$\forall(P : \text{is_semantic_predicate?}(P)) : \left(\begin{array}{l} \exists(\text{decider} : \text{computable}) : \\ \forall(pvs_0 : \text{partial_recursive}) : \\ (\neg\gamma(\text{decider})(\kappa_p(pvs_0), 0)) \Leftrightarrow P(pvs_0) \end{array} \right) \Leftrightarrow (P = \text{fullset} \vee P = \emptyset)$$

Proof Necessity: Suppose that $P = \text{fullset}$. Let \top be an element different from 0. The PVS0 program $\text{decider} = (\mathbf{O}_1, \mathbf{O}_2, 0, [\text{cnst}(\top)])$, decides fullset . Now, suppose that $P = \emptyset$. The PVS0 program $\text{decider} = (\mathbf{O}_1, \mathbf{O}_2, 0, [\text{cnst}(0)])$ decides \emptyset .

Sufficiency: by contraposition, let assume that $(P \neq \text{fullset} \wedge P \neq \emptyset)$. This implies that there exist PVS0 programs, say p and np , such that $P(p)$ and $\neg P(np)$. For reaching a contradiction, suppose that there exists $\text{decider} : \text{computable}$ such that:

$$\forall(pvs_0 : \text{partial_recursive}) : \neg\gamma(\text{decider})(\kappa_p(pvs_0), 0) \Leftrightarrow P(pvs_0)$$

And, consider the program *opp* with kernel:

$$\begin{aligned} opp = & [\text{ite}(\text{rec}(1, \text{rec}(1 + |decider'4| + |np'4| + |p'4|, \mathbf{vr})), \\ & \text{rec}(1 + |decider'4|, \mathbf{vr}), \\ & \text{rec}(1 + |decider'4| + |np'4|, \mathbf{vr}))] :: \\ & \text{map}_{(-+1)}(decider'4) :: \\ & \text{map}_{(-+1 + |decider'4|)}(np'4) :: \\ & \text{map}_{(-+1 + |decider'4| + |np'4|)}(p'4) \end{aligned}$$

Using the theorem 1 that there are programs in the model that can print their own Gödel number, making $E_f = opp$:

$$\begin{aligned} \exists(\text{print} : \text{partial_recursive}) : \\ \text{LET } self = (\mathbf{O}_1, \mathbf{O}_2, 0, opp :: \text{map}_{(-+|opp|)}(\text{print}'4)) \text{ IN} \\ \text{partial_recursive?}(self) \wedge \\ \forall(i) : \varepsilon(self)(\text{print}'4(0)^{+|opp|}, i, \kappa_p(self)) \end{aligned}$$

To understand how *opp* and *self* work, suppose that for each PVS0 program `partial_recursive` there is a function with the same name that executes the same as these. For example, for the PVS0 program denoted as “decider”, there is the corresponding function from naturals to naturals, also represented as “decider”. The same happens to the *p* and *np* PVS0 programs. The idea of the proof is to show a PVS0 program *self* that performs the same as the function:

$$self(n) := \text{IF } decider(\kappa_p(self)) \neq 0 \text{ THEN } np(n); \text{ ELSE } p(n);$$

The proof uses Cantor’s diagonal argument. If $decider(\kappa_p(self)) \neq 0$, then $P(self)$, but *self* behaves as *np* and thus $\neg P(self)$ holds, which is a contradiction. Otherwise, if $decider(\kappa_p(self)) = 0$, then $\neg P(self)$, but *self* behaves as *p* and thus $P(self)$ that is a contradiction too. This is the main idea behind the rest of the explanation of the formalization.

The theorem above 1 implies that there exists an element of the partial recursive class, say *print*, such that:

$$\begin{aligned} \text{LET } self = (\mathbf{O}_1, \mathbf{O}_2, 0, opp :: \text{map}_{(-+|opp|)}(\text{print}'4)) \text{ IN} \\ \text{partial_recursive?}(self) \wedge \\ \forall(i) : \varepsilon(self)(\text{print}'4(0)^{+|opp|}, i, \kappa_p(self)) \end{aligned}$$

Making $pvs_0 = self$ it can be concluded that

$$\neg\gamma(decider)(\kappa_p(self), 0) \Leftrightarrow P(self)$$

The proof splits into two sub-cases.

Sub-case 1: $P(self)$. In this case, $\neg\gamma(decider)(\kappa_p(self), 0)$ is concluded.

Since P is a semantic predicate, one has:

$$\begin{aligned} \forall(pvs_{01}, pvs_{02}) : \\ (\forall(i, o) : \gamma(pvs_{01})(i, o) \Leftrightarrow \gamma(pvs_{02})(i, o)) \Rightarrow \\ (P(pvs_{01}) \Leftrightarrow P(pvs_{02})) \end{aligned}$$

Thus, choosing pvs_{01} as *self* and pvs_{02} as *np*, it gives:

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

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

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

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

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

$$\varepsilon(\mathit{self})(\mathit{opp} :: \mathit{map}_{(-^{+|\mathit{opp}|})}(\mathit{print}'4)(0), i, o)$$

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

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

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

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

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

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

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

$$\begin{aligned} i = v''' \wedge \\ \varepsilon(\mathit{self})(\mathit{print}'4(0)^{+|\mathit{opp}|}, v''', v'') \wedge \\ \varepsilon(\mathit{self})(\mathit{decider}'4(0)^{+1}, v'', v') \wedge \\ \mathbf{IF } v' \neq \perp \mathbf{ THEN } \varepsilon(\mathit{self})(\mathbf{rec}(1 + |\mathit{decider}'4|, \mathbf{vr}), i, o); \\ \mathbf{ ELSE } \varepsilon(\mathit{self})(\mathbf{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \mathbf{vr}), i, o); \end{aligned}$$

By the second part of the aforementioned theorem 1, i.e., $\forall(i) : \varepsilon(\mathit{self})(\mathit{print}'4(0)^{+|\mathit{opp}|}, i, \kappa_p(\mathit{self}))$, and instantiating $i = v'''$ one obtains:

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

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

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

By using the shift code lemma (Lemma 1),

$$\begin{aligned}
& \varepsilon(\mathit{self})(\mathit{decider}'4(0)^{+1}, \kappa_p(\mathit{self}), v') \Leftrightarrow \\
& \varepsilon(\mathit{decider})(\mathit{decider}'4(0), \kappa_p(\mathit{self}), v')
\end{aligned}$$

Thus one obtains,

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

By the hypothesis of this case, one has $\neg\gamma(\mathit{decider})(\kappa_p(\mathit{self}), 0)$ that means that $v' \neq 0$. Thus,

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

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 the shift code lemma (Lemma 2):

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

which is equivalent to $\gamma(\mathit{np})(i, o)$. Thus one has that $\neg\forall(i, o) : \gamma(\mathit{self})(i, o) \Leftrightarrow \gamma(\mathit{np})(i, o)$ does not hold, which is a contradiction.

Sub-case 2: $\neg P(\mathit{self})$. It follows analogously to sub-case 1, except for the supposition that P is a semantic predicate where pvs_{0_1} and pvs_{0_2} are instantiated respectively as self and p , which leads to a contradiction.

6.2 Applications of Rice's Theorem

The generality of Rice's Theorem allows simple formalizations of significant 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 1 (Undecidability of the Uniform Halting Problem - `uniform_halting_problem_undecidability_Turing_complete` [↗](#))

$$\begin{aligned} &\neg\exists(\text{decider} : \text{computable}) : \\ &\quad \forall(\text{pvs}_0 : \text{partial_recursive}) : \\ &\quad\quad (\neg\gamma(\text{decider})(\kappa_p(\text{pvs}_0), 0) \Leftrightarrow T_\varepsilon(\text{pvs}_0)) \end{aligned}$$

Proof The formalization uses Rice's Theorem instantiating the semantic predicate as T_ε . The predicate T_ε is a semantic predicate because if two PVS0 programs perform the same, both are either terminating or not. Since the set T_ε is neither equal to the empty set nor the whole set `partial_recursive`, there exists no computable decider for this set. To prove this, it is shown that the `partial_recursive` constant program $(\mathbf{O}_1, \mathbf{O}_2, 0, [\text{cnst}(0)])$ belongs to T_ε , while a simple loop `partial_recursive` program specified as $(\mathbf{O}_1, \mathbf{O}_2, 0, [\text{rec}(0, \text{vr})])$ does not.

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

The PVS theory complementing this paper includes both the formalization of the corollary above and a direct formalization of the undecidability of the (Specific) Halting Problem for the multiple-function PVS0 model in the spirit of [7].

Corollary 2 (Undecidability of Existence of Fixed Points - `fixed_point_existence_undecidability_Turing_complete` [↗](#))

$$\begin{aligned} &\neg\exists(\text{decider} : \text{computable}) : \\ &\quad \forall(\text{pvs}_0 : \text{partial_recursive}) : \\ &\quad\quad (\neg\gamma(\text{decider})(\kappa_p(\text{pvs}_0), 0) \Leftrightarrow \exists(p) : \gamma(\text{pvs}_0)(p, p)) \end{aligned}$$

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

$$\lambda(\text{pvs}_0 : \text{partial_recursive}) : \exists(p) : \gamma(\text{pvs}_0)(p, p)$$

It is a semantic predicate because if two PVS0 programs perform the same either both contain a fixed point or neither do. The predicate is then shown to be different from the empty set and from the whole set `partial_recursive`. Indeed, on one side, the predicate holds for the program $(\mathbf{O}_1, \mathbf{O}_2, 0, [\text{cnst}(0)])$, showing that it is different from empty set. On the other side, it does not hold for the program $(\mathbf{O}_1, \mathbf{O}_2, 0, [\text{op2}(i, \text{vr}, \text{cnst}(1))])$ that performs the same as $\lambda(n : \mathbb{N}) : \kappa_2(n, 1)$, concluding that the predicate is not equal to `partial_recursive`.

Corollary 3 (Undecidability of Self Replication - self_replication_undecidability_Turing_complete [↗](#))

$$\begin{aligned} & \neg \exists (\text{decider} : \text{computable}) : \\ & \quad \forall (\text{pvs}_0 : \text{partial_recursive}) : \\ & \quad \quad (\neg \gamma(\text{decider})(\kappa_p(\text{pvs}_0), 0) \Leftrightarrow \\ & \quad \quad \quad \exists (p : \text{partial_recursive}) : \\ & \quad \quad \quad \quad \forall (i) : \gamma(p)(v_i, \kappa_p(p)) \wedge \gamma(\text{pvs}_0)(v_i, \kappa_p(p))) \end{aligned}$$

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

$$\begin{aligned} & \lambda (\text{pvs}_0 : \text{partial_recursive}) : \\ & \quad \exists (p : \text{partial_recursive}) : \\ & \quad \quad \forall (i) : \gamma(p)(i, \kappa_p(p)) \wedge \gamma(\text{pvs}_0)(i, \kappa_p(p)) \end{aligned}$$

The predicate above is a semantic predicate because if two PVS0 programs perform the same, either both return a Gödel number of a program that self-replicates or neither do.

The next step is showing that the predicate is neither the empty set nor the full `partial_recursive` set. Using the assumption of the Recursion Theorem and instantiating it with `[rec(1, vr)]`, one shows that the predicate is not empty. On the other side, the program `(O1, O2, 0, [op2(i, cnst(1), vr)])` shows that the predicate is not the whole `partial_recursive` set.

Corollary 4 (Undecidability of Functional Equivalence - pvs0_program_equivalence_undecidability_Turing_complete [↗](#))

$$\begin{aligned} & \neg \exists (\text{decider} : \text{computable}) : \\ & \quad \forall (\text{pvs}_{0_0}, \text{pvs}_{0_1} : \text{partial_recursive}) : \\ & \quad \quad (\neg \gamma(\text{decider})(\kappa_2(\kappa_p(\text{pvs}_{0_0}), \kappa_p(\text{pvs}_{0_1})), 0) \Leftrightarrow \\ & \quad \quad \quad (\forall (v_i, v_o) : \gamma(\text{pvs}_{0_0})(v_i, v_o) \Leftrightarrow \gamma(\text{pvs}_{0_1})(v_i, v_o)))) \end{aligned}$$

Proof Suppose that there exists a `computable` program `decider` that decides the above equivalence between PVS0 `partial_recursive` programs. Then, instantiate `pvs0_0` above as the constant zero program, `(O1, O2, 0, [cnst(0)])` simplifying in this manner the problem to decide whether a program performs the same as the constant zero program. The next step is instantiating Rice's Theorem (Theorem 2) with the predicate below.

$$\begin{aligned} & \lambda (\text{pvs}_0 : \text{partial_recursive}) : \\ & \quad \forall (v_i, v_o) : \\ & \quad \quad \gamma(\mathbf{O}_1, \mathbf{O}_2, 0, [\text{cnst}(0)])(v_i, v_o) \Leftrightarrow \gamma(\text{pvs}_0)(v_i, v_o) \end{aligned}$$

Indeed, the predicate above is a semantic predicate because either two PVS0 programs always return zero or not.

To prove that the predicate neither is the empty nor the full `partial_recursive` set, it is enough to show that the constant zero and one programs respectively belongs and does not to the predicate. After that, one uses the assumed program `decider` to build another program for deciding the equivalence to the constant zero program; this program is built as `(O1, O2, 0, [rec(1, op2(i, κp(O1, O2, 0, [cnst(0)]), vr)]) :: decider'4`, where `i` is the

index of the κ_2 function. Indeed, by using the shifting code lemmas, this program can be adapted to decide the equivalence with 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 the shifting code lemmas and by showing that the program built above is also terminating.

7 Other Formalized Results and Related Work

7.1 Other Formalized Results

As mentioned in the introduction, the development includes proofs of other results, such as the Undecidability of the Halting Problem. This theorem was formalized following the classical diagonalization method. The formalization starts supposing that there exists a partial recursive PVS0 program, called *oracle*, that decides if another partial recursive PVS0 program halts for a given input. This assumption will give rise to a contradiction. The input given to the program *oracle* is a natural that encodes a pair consisting of a PVS0 program and a natural. The encoding assumes an arbitrary Gödelization to transform the PVS0 program into a natural and the bijection from pairs of naturals to naturals to obtain the natural codifying the pair. The contradiction comes building a partial recursive PVS0 program, called *liar*, such that if *oracle* returns true for the input pair $\kappa_2(n, n)$, *liar* executes an infinite loop; otherwise, it returns the encoded pair. Running *liar* having the Gödel number of *liar* as the input, if *oracle* returns that *liar* halts, then it does not halt, but if *oracle* returns that it does not halt, then it halts.

Note that above it was only necessary to assume an arbitrary Gödelization; indeed, the contradiction to conclude the Halting Problem's undecidability depends only on using the Gödel number attributed to the *liar* program. This assumption contrasts with the specific Gödelization κ_p (Section 3) applied in the formalization of Recursion and Rice's Theorems. The construction of such a κ_p was necessary since in the formalization of the Recursion theorem one needs to implement computations of the Gödelization using PVS0 programs. Besides that, Rice's Theorem, as a consequence of the Recursion Theorem, also needs the same PVS0 implementations.

If the implementation of a specific Gödelization (using a PVS0 program) is required, then supply specific built-in operators is necessary. In contrast, if one needs assuming an arbitrary Gödelization only, then there is not required to provide built-in operators. Here, it is necessary to stress that the class of PVS0 programs with an empty set of built-in operators does not allow the implementation of a Gödelization. Also, for this class of PVS0 programs, the Halting Problem is decidable, but not using the same class of PVS0 programs for the oracles. Thus, a relevant observation about the Halting Problem happens if the sets of built-in operators are empty. In this case, the evaluation of `op1` and `op2` always will result in \perp , non terminating PVS0 programs still exist, and as a consequence of our formalization of this theorem, the Halting Problem for this class of PVS0 programs (with themselves) is also undecidable.

The undecidability of the Halting Problem for the multiple-function PVS0 model is specified as the theorem below.

Theorem 3 (Undecidability of the Halting Problem for PVS0 - `mf_pvs0_halting_problem_undecidability` [↗](#)) For all O_1, O_2, \perp , and κ_p , there is no program oracle of type `computable` such that for all $pvs_0 = (O_1, O_2, \perp, E_f)$ of type `partial_recursive` and for all $n \in \mathbb{N}$,

$$T_\varepsilon(pvs_0, n) \text{ if and only if } \neg\gamma(\text{oracle})(\kappa_2(\kappa_p(pvs_0), n), \perp).$$

This specification states the non-existence of an *oracle*, such that it does not return \perp (i.e., it returns true) for an encoded PVS0 program pvs_0 , together with an arbitrary input natural n if and only if pvs_0 halts for n . That means that no PVS0 program can decide if any pvs_0 halts for an input n .

Another exciting result formalized in this development is The Fixed-Point Theorem. It states that for any program f that transforms a program into another one, there exists p such that $f(p)$ performs the same as p . In the case of partial recursive PVS0 programs, the program f receives the Gödel number of p and returns another Gödel number. The PVS theory for the Fixed-Point Theorem has as arguments basic built-in operators such that, for the formalization, it must be possible to implement the universal partial recursive PVS0 program. Using these operators it also must be possible to build a PVS0 program such that it receives a natural as an argument, and split it into another two arguments, a and b . The natural a is a Gödel number of a PVS0 program applied to the own a , resulting in another Gödel number of another program applied to b . This last PVS0 program is called *diagonal*. Thus, the formalization consists in building the PVS0 program p in the following way: the Gödel number of p is a result of the program *diagonal* applied to the Gödel number of the program f composed with *diagonal*. Notice that in this formalization, transformations of Gödel numbers into programs and programs into Gödel numbers are required. This implies that the Gödelization function must have right and left inverses, i. e., it must be bijective.

The specification in PVS of the Fixed-Point Theorem is given below.

Theorem 4 (Fixed-Point Theorem for PVS0 - `fixed_point` [↗](#)) Assume that it is possible to build the universal and the diagonal PVS0 programs as described above using the built-in operators. Then, for all f of type `computable`, there exists p of type `partial_recursive`, such that for all v_i, v_{o_1} and v_{o_2} , input and outputs,

$$\gamma(p)(v_i, v_{o_1}) \wedge \gamma(\Delta(f)(p))(v_i, v_{o_2}) \Rightarrow v_{o_1} = v_{o_2}$$

where, Δ is a function that receives the Gödel number of p , applies the PVS0 program f resulting in a natural that is transformed in another PVS0 program.

In the specification above, p and $\Delta(f)(p)$ compute the same output for a given input. The chosen p is built as $\Delta(\text{diagonal})(f \circ \text{diagonal})$, where *diagonal* is as described in the previous paragraph.

7.2 Related work

Nowadays, mechanical proofs of computability properties are not only of interest as exercises of formalization but also of great importance to provide formal support to practical computational models. As mentioned in the introduction, the main aim of the single- and multiple-function PVS0 models is related to the development

of automation mechanisms to verify termination of PVS programs [1]. In [7], **PVS0** programs consist of a single function. Also, they are constrained in inductive levels such that in the level zero only the basic functions `successor`, `greater-than` and `projections` were 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 the formalization of results such as Turing completeness and Rice's Theorem difficult. As seen in Section 2, the composition of programs specified in the multiple-function **PVS0** language is straightforwardly achieved by the application of the offset operator `_+-`.

For the single-function **PVS0** language, the composition of two (not necessarily terminating) programs require the construction of a third program that cannot be specified in a general manner since this depends on the (combinatorial) structure of the input programs. In the proof of the undecidability of the Halting Problem in [7], 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 a model a language which supports the specification of programs that consist of several functions that can call not only themselves recursively, but that can also call each other.

The equivalencies between termination criteria were formalized for the single-function **PVS0** model considered in [7]. However, such equivalences have not been formalized for the current multiple-function **PVS0** model. Theoretically, all termination criteria mentioned in the introduction (references [20], [21], [4], [28], [3]) work for both models. Still, technically, some of the termination 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 discussed in [5], where the LISP language is the model of computation. A second example is the formalization in Agda, of the same undecidability result, reported in [16], where the used model of computation is built as axioms over the elements of an abstract type `Prog`.

Here, the focus is on recent works in which computability results have been formalized over such computational models related to lambda calculus and programming languages. In [12], Forster and Smolka used as a model of computation call-by-value lambda calculus, which is a Turing Complete model of computation, where beta-reduction can be applied only to a beta-redex that is not below an abstraction, and whose argument is an abstraction. For this model, the authors formalized several computational properties includes Rice's Theorem; indeed, they formalized that semantic predicates such that there are elements both in them and in their complements are not recognizable. This property is called Rice's Lemma by the authors, and it is used to conclude Rice's Theorem. Also, Norrish formalized in [22], 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. Rice's Theorem was also formalized by Carneiro in Lean [6] using partial recursive functions as a model of computation. And, the proof uses

the Fixed-Point Theorem to conclude Rice’s Theorem as a corollary; also, in this work, the undecidability of the Uniform Halting Problem is obtained as a corollary.

Notice that a difference among Carneiro’s [6], Norrish’s [22], Forster and Smolka’s [12] works and the current work is the model of computation. The resources given by the lambda-calculus allow a more straightforward formalization of Rice’s Theorem because it facilitates the implementation of operators necessary for the proof, for example, a fixed-point operator. In the formalization of Rice’s Theorem for the PVS0 model, the Recursion Theorem plays the fixed point construction role. Besides that, lambda-calculus does not need the Gödelization technique as required for the PVS0 model. Also, using partial recursive functions or a concrete functional language model such as PVS0 programs allows formalizations of properties of functional programs such as those related to their complexity and termination criteria, the latter objective that motivated the specification of the PVS0 language.

As previously discussed, the targetted computation model influences the selection of problems used to build reductions and the formalizations’ complexity. For example, the formalization of the Word Problem’s undecidability over monoids seems natural using Turing Machines as in the classical Post’s approach ([23]). It builds a reduction from the Halting Problem (over Turing Machines to the Word Problem). Related formalization approaches have been reported by several authors for other word problems such as PCP (e.g., [9], [14]). Post’s approach represents Turing Machines’ configurations as words and the action of transitions as reductions by a semi-Thue system. Atomicity of Turing Machine transitions allows representing each transition as a rewriting rule. Proving the Word Problem over the PVS0 model is much more elaborated since reducing the Halting Problem of PVS0 programs to the Word Problem is not straightforward. Nonatomicity of the evaluation steps of PVS0 programs makes difficult the construction of such a reduction. For example, the different “configurations” in the evaluation of $\kappa_2^S(\text{cnst}(4), \text{cnst}(7))$, using the built-in operators fixed in Section 3 would require a representation of the constants and, the symbol κ_2^S as a word (over some alphabet). Besides that, it requires a semi-Thue system that simulates the evaluation of κ_2 , which involves evaluating other operators as addition, multiplications, and division by two. In general, the complexity of proofs of undecidability properties depends on the selected computational model. The unique such choice in our formalization refers to the reduction from the Recursion to Rice’s Theorem. Despite the fact that there are well-known properties used in textbooks’ proofs, such as the assumption of the existence of a universal machine or the undecidability of the universal language (e.g., [24], [15]), to be best of our knowledge, complete formalizations of computational properties do not follow from such constructions. Instead, other strategies are followed, such as reductions from the Fixed-Point Theorem in [6], the Rice’s Lemma in [12] and the Recursion Theorem in the current work.

For the PVS0 model, the most straightforward manner to formalize the Rice’s Theorem was as a corollary of the Recursion Theorem using as basic built-in operators κ_2 , successor, the projections composed to κ_2^{-1} and greater-than. But note that for a Turing complete model as PVS0, the Fixed-Point and Recursion Theorem are equivalent being possible to prove each one from the other.

There are other exciting computability results formalized over linguistic computational models. In [10], Forster, Kirsk, and Smolka formalized 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. In [11], Forster and Larchey-Wendling formalized in Coq the reduction of the Post Correspondence Problem (PCP), via binary stack machines and Minsky machines, to provability of intuitionistic linear logic. They started with the PCP, and built a chain of reductions: passing through binary PCP, binary PCP with indices, binary stack machines, Minsky machines, and finally, provability of intuitionistic linear logic. In [18], Larchey-Wendling formalized that the type $\mathbb{N}^k \rightarrow \mathbb{N}$ in Coq contains every k -ary recursive function which can be proved total in Coq; this set of functions includes the class of primitive recursive functions. Trying to representing the class of partial recursive functions in Coq, for instance, using the type $\mathbb{N}^k \rightarrow \mathbf{option} \mathbb{N}$ would not work. The PVS version of the functor \mathbf{option} is the operator \mathbf{Maybe} used to add \diamond to the working type in the definition of the function of semantic evaluation χ (see Table 2). The attempt to use the type $\mathbb{N}^k \rightarrow \mathbf{option} \mathbb{N}$ to represent non-necessarily total functions in Coq fails because decidability of totality will contradict the undecidability of the Halting Problem. Therefore, the way to deal with this issue in Coq would be similar to the one used by the function χ of semantic evaluation of PVS0 programs, i.e., representing the partial recursive functions by a predicate of type $\mathbb{N}^k \rightarrow \mathbb{N} \rightarrow \mathbf{Prop}$.

Functional types in PVS allow specification of non-recursive functions. For example, the PVS function $non_comp(n)$ below has type $\mathbb{N} \rightarrow \mathbb{N}$. $non_comp(n) := max(\{v_o : \mathbb{N} \mid \gamma(\kappa_e^{-1}(\kappa_2^{-1}(n)'1))(\kappa_2^{-1}(n)'2, v_o)\} \cup \{-1\}) + 1$. The function receives a natural n that represents a PVS0 program and an input (respectively, $\kappa_e^{-1}(\kappa_2^{-1}(n)'1)$ and $\kappa_2^{-1}(n)'2$). The function is well-defined in PVS since maximum of finite sets is well-defined, and by the determinism of γ (see Section 2) the set $\{v_o : \mathbb{N} \mid \gamma(\kappa_e^{-1}(\kappa_2^{-1}(n)'1))(\kappa_2^{-1}(n)'2, v_o)\}$ is either empty or unitary. This is the reason why the built-in operators of the PVS0 model must be adequately fixed. This restriction avoids having a model over a set of non-recursive operators that will trivially be Turing Complete but also allow the specification of non-computable functions.

Recently in [19], the Larchey-Wendling and Forest formalized the undecidability of Hilbert’s Tenth Problem using a chain of reductions of problems: Halting Problem for TMs, PCP, a specialized Halting Problem for Minsky Machines, FRACTAN (a language model that deals with register machines) termination, and solvability of Diophantine logic and Diophantine equations.

More recently, Spies and Forster ([25]), and Kirst and Larchey-Wendling ([17]) added two interesting results to the Coq library of synthetic undecidability proofs; namely, the formalization of the undecidability of higher-order unification, and the undecidability of first-order satisfiability by finite models (FSAT). The former formalizes Goldfarb’s proof of the undecidability of second-order unification by a reduction from Hilbert’s Tenth problem [13], from which the general result for higher-order unification is a corollary. The latter result is known as Trakhtenbrot’s Theorem, originally proved by a reduction from the Halting Problem for Turing machines [26]. The formalization in [17] is obtained by a reduction from PCP. It includes a sharper version of the undecidability of FSAT for signatures that contain either an at least binary relation symbol or a unary relation symbol together with an at least binary function symbol.

8 Conclusions and Future Work

The functional language *PVS0* is formally studied as a model of computation. Turing completeness of *PVS0* was formalized in the proof assistant PVS for a subclass of so-called partial recursive *PVS0* programs over the type of naturals and built from basic operators for successor, projection, and greater-than functions and bijective operators to encode tuples from naturals and vice versa. The proof consists of formalizations of the correctness of *PVS0* implementations of these functions and operators for composition, primitive recurrence, and minimization.

Additionally, Rice’s Theorem is formalized in PVS for the *PVS0* model. The proof uses the Recursion Theorem and a Gödelization of *PVS0* programs and follows Cantor’s diagonal argument to build a contradiction arising from the existence of a *PVS0* program that can decide semantic predicates about *PVS0* programs. Applications of Rice’s Theorem include formalizations of corollaries such as undecidability of the uniform Halting Problem, the functional equivalence problem, the existence of fixed points problem, and self-replication. The development also includes formalizations of the undecidability of the Halting Problem and Fixed-Point Theorem for *PVS0*.

This part of the *PVS0* development added 273 proved lemmas from which 177 are *Type Correctness Conditions* (TCCs) that are proof obligations automatically generated (but not necessarily proved) by PVS. The quantitative data of the files of proofs in Figure 1, is given in Table 3. Data of other auxiliary theories that did not require a substantial amount of work (in the totals above) are not included in the table.

Table 3 Relevant quantitative data

PVS theory	Lines of Code (loc) and Size of proof files	Proved formulas	Proved TCCs
<code>mf_pvs0.Rices.Theorem</code>	5206 loc - 594K	2	2
<code>mf_pvs0.Recursion.Theorem</code>	6754 loc - 572K	7	14
<code>mf_pvs0.Turing.Completeness</code>	17677 loc - 1,5M	27	32
<code>mf_pvs0.Fixedpoint</code>	4712 loc - 68K	1	4
<code>mf_pvs0.Halting</code>	1704 loc - 149K	2	3
<code>mf_pvs0.Rices.Theorem.Corollaries</code>	1786 loc - 100K	5	0
<code>mf_pvs0.basic.programs</code>	4879 loc - 319K	9	10
<code>mf_pvs0.computable</code>	6586 loc - 310K	9	50
<code>mf_pvs0.lang</code>	2817 loc - 122K	20	13
<code>mf_pvs0.expr</code>	3418 loc - 166K	7	47

Although the size of proofs for Turing Completeness doubles the size of proofs of the Recursion Theorem, the former formalization is simpler; indeed, several auxiliary proofs that are applied to formalize Turing Completeness are related to technical and simple issues for which semantic evaluation is required (i.e., expansions of the definition of the predicate ε) and simple instantiations of premises existentially quantified. Also, some of the auxiliary theories require a substantial number of lemmas; indeed, the theories `mf_pvs0.expr` and `mf_pvs0.lang` include several proofs related to the correctness of the operational semantics of the multiple-function *PVS0* language and, the theory `mf_pvs0.computable` includes all results related with Gödelizations.

Other results of interest to be formalized for the *PVS0* language model are the s - m - n theorem, the undecidability of PCP, Post's Theorem, the existence of a universal machine, the existence of self-replicating 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 ([23]) and Hilbert's Tenth Problem [19].

Recent examples of related developments include the formalizations in Coq of the Post's theorem for weak call-by-value lambda-calculus [12] and of the undecidability of the PCP via reduction of the Halting Problem for Turing machines [9]. 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.

Current work includes the formalization of the undecidability of PCP. This is important to deal with the undecidability of problems outside of the field of computability such as the Word Problem over algebraic structures and, SAT (as done in [17] for FSAT). Also, as mentioned in the section on related work, providing translations from the multiple- to the single-function *PVS0* language would be of great interest to check termination properties of multiple-function *PVS0* programs.

References

1. Alves Almeida, A., Ayala-Rincon, M.: Formalizing the Dependency Pair Criterion for Innermost Termination. *Science of Computer Programming* **195**(102474) (2020). URL <https://doi.org/10.1016/j.scico.2020.102474>
2. Arts, T.: Termination by Absence of Infinite Chains of Dependency Pairs. In: 21st International Colloquium on Trees in Algebra and Programming CAAP, LNCS, vol. 1059, pp. 196–210. Springer (1996). URL https://doi.org/10.1007/3-540-61064-2_38
3. Arts, T., Giesl, J.: Termination of term rewriting using Dependency Pairs. *Theoretical Computer Science* **236**, 133–178 (2000). URL [https://doi.org/10.1016/S0304-3975\(99\)00207-8](https://doi.org/10.1016/S0304-3975(99)00207-8)
4. Avelar, A.B.: Formalização da Automação da Terminação Através de Grafos com Matrizes de Medida. Ph.D. thesis, Department of Mathematics, Universidade de Brasília (2014). URL <http://repositorio.unb.br/handle/10482/18069>. In Portuguese
5. Boyer, R.S., Moore, J.S.: A Mechanical Proof of the Unsolvability of the Halting Problem. *Journal of the Association for Computing Machinery* **31**(3), 441–458 (1984). URL <https://doi.org/10.1145/828.1882>
6. Carneiro, M.: Formalizing Computability Theory via Partial Recursive Functions. In: 10th International Conference on Interactive Theorem Proving ITP, LIPICs, vol. 141, pp. 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). URL <https://doi.org/10.4230/LIPICs.ITP.2019.12>
7. Ferreira Ramos, T.M., Muñoz, C.A., Ayala-Rincón, M., Moscato, M.M., Dutle, A., Narkawicz, A.: Formalization of the Undecidability of the Halting Problem for a Functional Language. In: 25th International Workshop on Logic, Language, Information, and Computation WoLLIC, LNCS, vol. 10944, pp. 196–209. Springer (2018). URL https://doi.org/10.1007/978-3-662-57669-4_11
8. Floyd, R.W., Beigel, R.: *The Language of Machines: An Introduction to Computability and Formal Languages*. W H Freeman & Co (1994). URL <https://doi.org/10.2307/2275690>

9. Forster, Y., Heiter, E., Smolka, G.: Verification of PCP-Related Computational Reductions in Coq. In: 9th International Conference on Interactive Theorem Proving ITP, *LNCS*, vol. 10895, pp. 253–269. Springer (2018). URL https://doi.org/10.1007/978-3-319-94821-8_15
10. Forster, Y., Kirst, D., Smolka, G.: On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem. In: 8th ACM SIGPLAN International Conference on Certified Programs and Proofs CPP, pp. 38–51. ACM (2019). URL <https://doi.org/10.1145/3293880.3294091>
11. Forster, Y., Larchey-Wendling, D.: Certified Undecidability of Intuitionistic Linear Logic via Binary Stack Machines and Minsky Machines. In: 8th ACM SIGPLAN International Conference on Certified Programs and Proofs CPP, pp. 104–117. ACM (2019). URL <https://doi.org/10.1145/3293880.3294096>
12. Forster, Y., Smolka, G.: Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In: 8th International Conference on Interactive Theorem Proving ITP, *LNCS*, vol. 10499, pp. 189–206. Springer (2017). URL https://doi.org/10.1007/978-3-319-66107-0_13
13. Goldfarb, W.D.: The undecidability of the second-order unification problem. *Theoretical Computer Science* **13**, 225–230 (1981). URL [https://doi.org/10.1016/0304-3975\(81\)90040-2](https://doi.org/10.1016/0304-3975(81)90040-2)
14. Heiter, E.: Undecidability of the Post Correspondence Problem. Master’s thesis, Faculty of Mathematics and Computer Science, Saarland University (2014). URL https://www.ps.uni-saarland.de/~heiter/downloads/PCP_Undecidability.pdf
15. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, third edn. Pearson (2008). URL <https://doi.org/10.1145/568438.568455>
16. Johannisson, K.: Formalizing the halting problem in a constructive type theory. In: International Workshop on Types for Proofs and Programs TYPES, *LNCS*, vol. 2277, pp. 145–159. Springer (2000). URL https://doi.org/10.1007/3-540-45842-5_10
17. Kirst, D., Larchey-Wendling, D.: Trakhtenbrot’s Theorem in Coq, A Constructive Approach to Finite Model Theory. *CoRR abs/2004.07390* (2020). URL <https://arxiv.org/abs/2004.07390>
18. Larchey-Wendling, D.: Typing Total Recursive Functions in Coq. In: 8th International Conference on Interactive Theorem Proving ITP, *LNCS*, vol. 10499, pp. 371–388. Springer (2017). URL https://doi.org/10.1007/978-3-319-66107-0_24
19. Larchey-Wendling, D., Forster, Y.: Hilbert’s Tenth Problem in Coq. In: 4th International Conference on Formal Structures for Computation and Deduction FSCD, *LIPICs*, vol. 131, pp. 27:1–27:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2019). URL <https://doi.org/10.4230/LIPICs.FSCD.2019.27>
20. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The Size-Change Principle for Program Termination. In: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 81–92 (2001). URL <http://doi.acm.org/10.1145/360204.360210>
21. Manolios, P., Vroon, D.: Termination Analysis with Calling Context Graphs. In: 18th International Conference on Computer Aided Verification CAV, *LNCS*, vol. 4144, pp. 401–414. Springer (2006). URL https://doi.org/10.1007/11817963_36
22. Norrish, M.: Mechanised Computability Theory. In: Second International Conference on Interactive Theorem Proving ITP, *LNCS*, vol. 6898, pp. 297–311. Springer (2011). URL https://doi.org/10.1007/978-3-642-22863-6_22
23. Post, E.L.: Recursive unsolvability of a problem of Thue. *The Journal of Symbolic Logic* **12**(1), 1–11 (1947). URL <https://doi.org/10.2307/2267170>
24. Sipser, M.: Introduction to the Theory of Computation, third edn. Cengage Learning (2012). URL <https://doi.org/10.1145/230514.571645>
25. Spies, S., Forster, Y.: Undecidability of higher-order unification formalised in Coq. In: 9th ACM SIGPLAN International Conference on Certified Programs and Proofs CPP, pp. 143–157. ACM (2020). URL <https://doi.org/10.1145/3372885.3373832>
26. Trakhtenbrot, B.A.: The impossibility of an algorithm for the decidability problem on finite classes. *Doklady Akademii Nauk SSSR* **70**(4), 569–572 (1950)
27. Turing, A.M.: Computability and λ -definability. *The Journal of Symbolic Logic* **2**(4), 153–163 (1937). URL <https://doi.org/10.2307/2268280>
28. Turing, A.M.: Checking a Large Routine. In: M. Campbell-Kelly (ed.) *The Early British Computer Conferences*, pp. 70–72. MIT Press, Cambridge, MA, USA (1989). URL <http://dl.acm.org/citation.cfm?id=94938.94952>