**Noname manuscript No.**
(will be inserted by the editor)

# 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 successor, projections, greater-than and bijections from tuples of naturals to naturals, results in a model that is proved (formalized) to be Turing complete. Complete formalizations of the Recursion Theorem and Rice's Theorem are discussed in detail. Other relevant results such as the undecidability of the Halting Problem, 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, which means that not all kinds of *problems* can be solved and that *programs* over each of these models with the same or different semantics cannot be distinguished. One of these limits, 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.

The formalization of undecidability results is presented in the proof assistant PVS, using as model of computation a variant of a first-order functional language called `PVS0`. In this language there are constants, only one variable, unary and binary built-in operators, if-then-else instructions and recursive calls. `PVS0` was designed with the main aim of formalizing the correctness of mechanisms to automate verification of termination in PVS, which makes of high interest the exploration of the computability theory of `PVS0` as a model of computation. Indeed,

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

PVS0 is applied (i.e. used as a model of computation) to formalize the equivalence among different criteria of termination, including size change principle ([19]) based termination criteria such as Manolios and Vroom's Calling Context Graphs [20] and Avelar *et al* Matrix Weighted Graphs [4], as well as Turing termination [27] (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`.

The aim of this paper is related to a second relevant objective that is the study of PVS0 as a model of computation, for which the formalization of its *computational theory* is required. In addition to verification of termination criteria over PVS0, it is important to formalize computability properties of the model to certify that it is indeed a reasonable and expressive model of computation. Previous work presented the formalization of the undecidability of the Halting Problem for the PVS0 model [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. The language model used in this paper is more realistic, accepting, similarly to functional specifications, a list of functions such that each function can call and be called by 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, but this result can also be obtained just as a corollary of the formalization of Rice's theorem.

The language PVS0 was designed to be similar to the specification language of PVS in order to allow a meta-theoretical study of the properties associated with PVS itself. Therefore, this study attempts not only a better understanding, but also eventual improvements of the PVS model. The differences between functions specified in PVS0 and in PVS are: in PVS the input and output types of a function need not be equal, while in PVS0 they should be equal since it works with a unique input/output type; PVS0 allows mutual recursion while PVS does not; and, the PVS grammar is much richer than the PVS0 grammar. The PVS0 grammar was chosen to be *minimal* in order to simplify formalizations; indeed, reducing the number of grammatical elements also reduces the number of cases to be considered in proofs of properties about 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 PVS theory also requires, as 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 that is 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 consists in proving 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. This follows the lines of proofs such as the one in [26] that shows $\lambda$-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 elaborated 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 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 the implementation of an elaborated PVS0 program that calculates its own Gödel number. The formalization follows the lines of proof as given in [23].
- Rice's Theorem. It was formalized as a corollary of the Recursion Theorem, which was used to build a partial recursive PVS0 program that processes its own Gödel number and, if this 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 [23] for Turing Machines.
- Additional results such as the undecidability of the Halting Problem and the Fixed-Point Theorem were also formalized. There are two versions of the Theorem of undecidability of the Halting Problem: one says that it is undecidable if a program halts for a specific input (Halting Problem) and 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 [23] for Turing Machines. The Fixed-Point Theorem was formalized as consequence of the fact that it is possible to build the universal PVS0 program and 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 result to the second argument. This formalization is the only one that requires to use 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.

To prove the Recursion Theorem it is necessary to build a partial recursive PVS0 program that computes its own Gödel number. This is done by dividing the

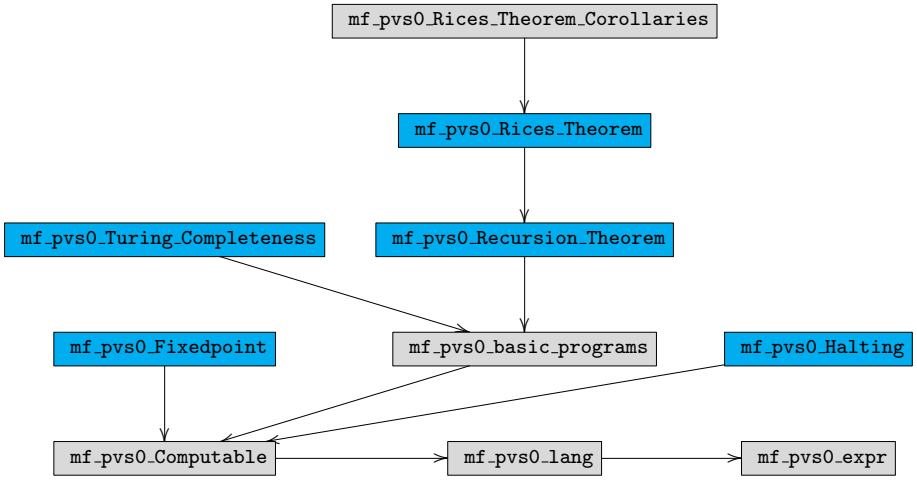


**Fig. 1** Hierarchy of the PVS0 theory related with this extension

expression list of the program 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 ([23]) for Turing Machines, where Turing Machines are used to print and process their own descriptions. In this work, Sipser's approach is adapted to build functional programs that output their own Gödel number. The main difficult is that Sipser's proof informally describes how to build such Turing Machines, while here, in order 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 separability of semantic properties of machines. The main differences with 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 undecidability of the Halting Problem, being concluded from the Recursion Theorem without using any translation to or from other computational models. Sipser's textbook [23] includes a 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.

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

In textbooks, assumptions such as the existence of universal machines and programs, 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. In order to rule out such kind of assumptions and get complete proofs, specific 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 is provided as part of this submission. It requires the installation of the NASA library https://github.com/nasa/pvslib, which includes the `PVS0` library for termination criteria and their equivalence over the single-function `PVS0` model.

## 2 Semantics of `PVS0` Programs

Expressions of the `PVS0` functional language have the grammar below.

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

Above, $T$ is an uninterpreted type over which `PVS0` expressions are interpreted. In the formalization the grammar is implemented as an abstract datatype (ADT) built from $T$. This type is the type of the input/output used in the evaluation of expressions (and programs). Constants of type $T$ are represented by the constructor $\texttt{cnst}(T)$ and the symbol `vr` is the unique symbol of variable. The main advantage of using such a simple grammar is generating a low number of cases to be analyzed in the proofs. `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 that are used to interpret operator symbols. The symbol `rec` is for function calls and uses also natural indices that are 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 ADT, PVS generates the required basic functions and axioms; for instance, the *subterm* relation is generated as below.

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

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 in the head of the list. Recursive calls, $\texttt{rec}(i, \_)$, are interpreted as evaluations (or function calls) of the $i^{th}$ PVS0 expression in the list. PVS0 programs include also lists of unary and binary functions to interpret the symbols of unary and binary operators, and an element of the 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 $(O_1, O_2, \bot, E_f)$, where $O_1$ and $O_2$ are the lists of unary and binary functions (specified in PVS), $\bot$ is an element of $T$, and $E_f$ is the kernel of the program. Note that the evaluation of the expression given as first argument of an `ite` symbol must also be an element of type $T$ interpreted as a Boolean. Thus, in order to guarantee that `ite` has the semantics of if-then-else instructions, the interpretation of false as an specific element $\bot$ of type `T` is necessary.

List of $n$-elements are denoted as $[a_0, .., a_{n-1}]$. $L(i)$ denotes the $i^{th}$ element of the list $L$, and $|L|$ its length. The tail of a non-empty list $L$ is denoted as $cdr(L)$, 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 $map(f)(L)$.

The (eager) evaluation predicate $\varepsilon$, for PVS0 programs is defined in Table 1.

Parameters $v_i$ and $v_0$ of the predicate $\varepsilon$ are the input and output values, and the parameter $e$ is the (sub)expression being evaluated of the kernel of the PVS0 program $(O_1, O_2, \bot, E_f)$, for short denoted as $pvso$. When evaluating this expression, its subexpressions can match with 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 $pvso$.

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

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

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

To prove properties related to automation of termination, semantic termination of PVS0 programs was specified in [7]. This definition is required to prove completeness and equivalence of practical termination criteria as well as to formalize computablity results such as undecidability of the Halting Problem and Rice's theorem. The *semantic termination predicate* is specified as below.

**Table 1** Evaluation predicate $\varepsilon$ for PVS0 programs

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

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

This predicate states that for a given program *pvso* and input $v_i$, the evaluation on the value $v_i$ terminates with the output value $v_o$. The program *pvso* is *total with respect to* $\varepsilon$ if it satisfies the following predicate (here notice that polymorphism in PVS allows the use of the same function or predicate name with different types).

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

Both the formalizations of the undecidability of the Halting Problem as given in [7] and Rice's Theorem in this paper require to build composition of functions and programs to give rise to contradictions. Using the multiple-function PVS0 model in this work, it is possible to specify the composition of arbitrary PVS0 functions, but for the single-function model used in [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 that share the same unary and binary operators and interpretation of false, a fundamental issue is the notion of *offset* used to adjust the indices of function calls in program lists. It works as a simplified version of offset in assembly languages, where instruction labels in some piece of code are adjusted when they are shifted. For PVS0 programs this is specified as the function $\beta$.

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

The function $\beta$ adds $n$ to all the `rec` indices in the expression $e$. Using $\beta$, the composition of two `PVS0` programs $(O_1, O_2, \perp, A)$ and $(O_1, O_2, \perp, B)$, for short $pvso_A$ and $pvso_B$, respectively, is expressed by the property:

$$\begin{aligned}
&\forall(v_i, v_o) : \exists(v) : \gamma(pvso_B)(v_i, v) \wedge \gamma(pvso_A)(v, v_o) \Leftrightarrow \\
&\gamma(O_1, O_2, \perp, [\texttt{rec}(1, \texttt{rec}(1 + |A|, \texttt{vr}))] :: map(\beta(1))(A) :: \\
&\qquad\qquad map(\beta(1 + |A|))(B))(v_i, v_o)
\end{aligned}$$

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.

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

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

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

The correctness of the composition of *factorial* and *quadratic* using $\beta$, is expressed as the property below.

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

Where, more concretely $map(\beta(1))(\mathit{factorial})$ is the expression $\texttt{ite}(\texttt{vr}, \texttt{op2}(0, \texttt{vr}, \texttt{rec}(1, \texttt{op1}(0, \texttt{vr}))), \texttt{cnst}(1))$ and $map(\beta(1 + |\mathit{factorial}|))(\mathit{quadratic})$ is $\texttt{op2}(0, \texttt{vr}, \texttt{vr})$; thus, the list of expressions of the composition is:

$$\begin{aligned}
&[\texttt{rec}(1, \texttt{rec}(2, \texttt{vr}))] :: \\
&[\texttt{ite}(\texttt{vr}, \texttt{op2}(0, \texttt{vr}, \texttt{rec}(1, \texttt{op1}(0, \texttt{vr}))), \texttt{cnst}(1))] :: \\
&[\texttt{op2}(0, \texttt{vr}, \texttt{vr})]
\end{aligned}$$

A functional alternative for the semantic evaluation predicate $\varepsilon$ 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 $\diamondsuit$. In addition, the evaluation function includes a parameter that limits the allowed number of nested recursive calls: when this limit is reached, the function returns $\diamondsuit$. This is given as the function $\chi$ in Table 2.

**Table 2** Evaluation function $\chi$ for PVS0 programs

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

The predicate $\varepsilon$ and function $\chi$ are proved equivalent in the following sense:

$$
\begin{array}{l}
\forall(pvso, e, v_i, v_o) : \varepsilon(pvso)(e, v_i, v_o) \Leftrightarrow \\
\qquad \exists(n) : \chi(pvso)(n, e, v_i) = v_o \wedge v_o \neq \diamondsuit
\end{array}
$$

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

Having two (equivalent) notions of operational semantics for the PVS0 language provides higher flexibility in the formalization since properties may be checked selecting one of these notions alternatively. In particular, the semantics provided by the function $\chi$ turns clear the measure required in inductive proofs; namely, to show termination of $\chi$, 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. And in general, this is also the measure used to prove 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:

$$
\begin{array}{l}
valid\_index\_rec(e, n) := \\
\quad \forall(i, e_1) : subterm(\texttt{rec}(i, e_1), e) \Rightarrow i < n \\
\\
valid\_index(E_f) := \\
\quad \forall(i < |E_f|) : valid\_index\_rec(E_f(i), |E_f|)
\end{array}
$$

## 3 PVS0 Computable and Partial Recursive Classes

Let $\boldsymbol{O_1}$, $\boldsymbol{O_2}$ and $\bot$ be fixed unary, binary operators and a natural number to be considered as false in a evaluation. Besides that, fix the input and output type as naturals. Below, a class of partial recursive functions is defined as follows:

$$\begin{aligned} \mathit{partial\_recursive}(\mathit{pvso}) \;:=\; & \mathit{pvso}'1 = \boldsymbol{O_1} \;\wedge\; \mathit{pvso}'2 = \boldsymbol{O_2} \;\wedge \\ & \mathit{pvso}'3 = \bot \;\wedge\; \mathit{valid\_index}(\mathit{pvso}'4) \end{aligned}$$

Computable *partial_recursive* functions are given as:

$$\mathit{computable}(\mathit{pvso}) := \mathit{partial\_recursive}(\mathit{pvso}) \;\wedge T_\varepsilon(\mathit{pvso})$$

Given a *pvso* program, if $\mathit{partial\_recursive}(\mathit{pvso})$ holds, *pvso* is of type `PartialRecursive`. If in addition *pvso* is terminating, i.e., $\mathit{computable}(\mathit{pvso})$ holds, it is of type `Computable`.

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

**Lemma 1 (Shift code)**

$$\begin{aligned} & \forall(O_1, O_2, \bot, A, B, e, v_i, n) \;:\; \chi(\mathit{pvso}_B)(n, e, v_i) = \\ & \chi(O_1, O_2, \bot, A :: \mathit{map}(\beta(|A|))(B))(n, \beta(|A|)(e), v_i) \end{aligned}$$

This lemma means that, in an evaluation of the expression $e$ considering the `PVS0` program $\mathit{pvso}_B$, 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)**

$$\begin{aligned} & \forall(O_1, O_2, \bot, B, v_i, n) : \\ & \forall(A \mid \mathit{valid\_index}(A)) : \forall(e \mid \mathit{valid\_index\_rec}(e, |A|)) : \\ & \quad \chi(\mathit{pvso}_A)(n, e, v_i) = \chi(O_1, O_2, \bot, A :: B)(n, e, v_i) \end{aligned}$$

This lemma is similar to Lemma 1, but the indices of the `rec` expressions in the evaluated expression $e$ and in the list $A$ of the `PVS0` program $\mathit{pvso}_A$ must be valid references to a `PVS0` expression in $A$, and the list $B$ of the `PVS0` program $\mathit{pvso}_B$ 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 $\mathbb{N} \times$ `PVS0Expr`, where `PVS0Expr` is the type of `PVS0` expressions. Previous lemmas and the equivalence of $\varepsilon$ and $\chi$ entails both:

$$\begin{aligned} & \forall(O_1, O_2, \bot, A, B, e, v_i, v_o) : \varepsilon(\mathit{pvso}_B)(e, v_i, v_o) \Leftrightarrow \\ & \varepsilon(O_1, O_2, \bot, A :: \mathit{map}(\beta(|A|))(B))(\beta(|A|)(e), v_i, v_o) \end{aligned}$$

and

$$\begin{aligned} & \forall(O_1, O_2, \bot, B, v_i, v_o) : \\ & \forall(A \mid \mathit{valid\_index}(A)) : \forall(e \mid \mathit{valid\_index\_rec}(e, |A|)) : \\ & \quad \varepsilon(\mathit{pvso}_A)(e, v_i, v_o) \Leftrightarrow \varepsilon(O_1, O_2, \bot, A :: B)(e, v_i, v_o) \end{aligned}$$

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 semantic predicate over `PVS0` programs is specified as:

$$\begin{aligned} &\mathit{is\_semantic\_predicate}(P) := \forall(\mathit{pvso}_1, \mathit{pvso}_2) : \\ &(\forall(v_i, v_o) : \gamma(\mathit{pvso}_1)(v_i, v_o) \Leftrightarrow \gamma(\mathit{pvso}_2)(v_i, v_o)) \Rightarrow \\ &\qquad\qquad (P(\mathit{pvso}_1) \Leftrightarrow P(\mathit{pvso}_2)) \end{aligned}$$

If the predicate $P$ is a semantic predicate then it is of the type `SemanticPredicate`. 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 ($\kappa_e$) is built over a bijection from pairs of naturals to naturals $\kappa_2$ as below.

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

$$\begin{aligned} \kappa_e(\mathit{length})(\mathit{expr}) := {} & \texttt{CASES } \mathit{expr} \texttt{ OF} \\ \texttt{vr} & : 0; \\ \texttt{cnst}(v) & : v \times 5 + 1; \\ \texttt{rec}(j, e_1) & : (j + \kappa_e(\mathit{length})(e_1) \times (\mathit{length} + 1)) \times 5 + 2; \\ \texttt{op1}(j, e_1) & : \kappa_2(j, \kappa_e(\mathit{length})(e_1)) \times 5 + 3; \\ \texttt{op2}(j, e_1, e_2) & : \kappa_2(j, \kappa_2(\kappa_e(\mathit{length})(e_1), \kappa_e(\mathit{length})(e_2))) \times 5 + 4; \\ \texttt{ite}(e_1, e_2, e_3) & : \kappa_2(\kappa_e(\mathit{length})(e_1), \kappa_2(\kappa_e(\mathit{length})(e_2), \kappa_e(\mathit{length})(e_3))) \times 5 + 5; \end{aligned}$$

In the function $\kappa_e$ above, for all subexpression $\texttt{rec}(i, e)$ of the argument *expr*, $i$ is less or equal than *length*. The reason for this is because the goal is to Gödelize `PartialRecursive` programs and 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 $\alpha$ was implemented as below.

$$\begin{aligned} \mathit{rdc}(l) &:= \mathit{reverse}(\mathit{cdr}(\mathit{reverse}(l))) \\ \alpha_{aux}(l) &:= \texttt{IF } |l| = 1 \texttt{ THEN } l(0); \\ &\qquad \texttt{ELSE } \kappa_2(\alpha_{aux}(\mathit{rdc}(l)), l(|l| - 1)) \\ \alpha(l) &:= \texttt{IF } |l| = 0 \texttt{ THEN } 0; \\ &\qquad \texttt{ELSE } \kappa_2(|l| - 1, \alpha_{aux}(l)) + 1 \end{aligned}$$

Above, *reverse* reverses lists and *rdc* deletes the last element of a non-empty list. Notice that $\alpha$, through applications of $\alpha_{aux}$, transforms recursively the prefix of the input list without the last element into a natural and applies the bijection $\kappa_2$ to this natural and the last element of the list. In the Gödelization, the function $\alpha_{aux}$ receives a non empty list of naturals each representing an expression in a list of `PVS0` expressions. The structure of this construction becomes similar to the ones previously used and ease the proof of the Recursion Theorem. In particular, it will be helpful when $\alpha$ is used in inductive proofs in which `PVS0` programs are

built adding to the kernel a constant that represents a number associated to the Gödelization of a list of expressions.

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

$$\kappa_p(pvs0) := \alpha(map(\kappa_e(|pvso'4| - 1))(pvso'4)) - 1$$

The function $\kappa_p$ Gödelizes the `PartialRecursive PVS0` programs.

To prove bijectivity of $\kappa_p$, it was necessary to build the inverses of $\kappa_2$, $\kappa_e$, $\alpha_{aux}$ (and $\alpha$) given respectively as $\kappa_2^{-1}$, $\kappa_e^{-1}$, $\alpha_{aux}^{-1}$ and $\alpha^{-1}$. 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 $\kappa_p$, but they do not use its bijectivity; any Gödelization function can be used. These inverses were specified as below.

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

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

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

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

In the function $\kappa_e^{-1}$, 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 inverse of $\kappa_p$ is specified as below.

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

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

**Lemma 3 (Left and Right inversibility of $\kappa_e$ and $\alpha_{aux}$)**

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 operators in $\boldsymbol{O_1}$ and $\boldsymbol{O_2}$ include the functions successor, projections, greater-than and the function $\kappa_2$. Projections are built using the inverse of the function $\kappa_2$ .

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

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

The following class of PVS0 programs is defined, passing these operators as parameters of the theory:

$$\begin{array}{l} partial\_recursive\_TC(pvso) := \\ \quad pvso'1 = [succ, \pi_1, \pi_2] \wedge \\ \quad pvso'2 = [greater, \kappa_2] \wedge \\ \quad pvso'3 = 0 \wedge \\ \quad valid\_index(pvso'4) \end{array}$$

Any PVS0 program *pvso* that belongs to the above predicate is of the type PartialRecursiveTC. 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 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 with the implementation of projection, and closedness of the class under composition, minimization and primitive recurrence.

It is important to note that, for a class of partial recursive PVS0 programs, the lists of unary and binary operators and the element considered as false are fixed. Thus, even though their implementations (which remains the same as in the given PVS0 program) are necessary to completely specify equality, projection, composition, minimization and primitive recurrence for PVS0 programs, for simplicity only the list of PVS0 expressions (i.e. *pvso*'4) will be treated in this paper.

The implementation of projection requires a representation of different $n$-tuples of natural numbers using naturals. The $n$-tuples in PVS are specified as lists of naturals. The representation is specified below.

$$\begin{array}{l} \quad nat2list(n, x) := \\ \texttt{IF}\ \ n = 0\ \texttt{THEN}\ [\,]; \\ \texttt{ELSIF}\ \ n = 1\ \texttt{THEN}\ [x]; \\ \texttt{ELSE}\ \ [\kappa_2^{-1}(x)'1] :: nat2list(n - 1, \kappa_2^{-1}(x)'2); \end{array}$$

The function *nat2list* transforms a natural $x$ into a list of length $n$.
The following abbreviations are used:

$$succ^S(e) := \texttt{op1}(0, e);\ \pi_1^S(e) := \texttt{op1}(1, e);\ \pi_2^S(e) := \texttt{op1}(2, e);$$
$$greater^S(e_1, e_2) := \texttt{op2}(0, e_1, e_2);$$
$$\kappa_2^S(e_1, e_2) := \texttt{op2}(1, e_1, e_2).$$

The `PVS0 PartialRecursiveTC` program *equal*, specified below, verifies if the naturals in a tuple codified as a unique natural are equal.

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

A projection uses the implementation, $proj\_aux$, specified below.

$$\begin{array}{l}
\texttt{LET}\ j = \pi_1^S(\texttt{vr}),\ i = \pi_1^S(\pi_2^S(\texttt{vr})),\ n = \pi_1^S(\pi_2^S(\pi_2^S(\texttt{vr}))), \\
\ x = \pi_2^S(\pi_2^S(\pi_2^S(\texttt{vr}))), k_4(a, b, c, d) = \kappa_2^S(a, \kappa_2^S(b, \kappa_2^S(c, d)))\texttt{IN} \\
proj_aux'4 := \\
[\texttt{ite}(greater(i, j), \\
\quad \texttt{rec}(0, k_4(succ^S(j), i, n, \pi_2^S(x))), \\
\quad \texttt{ite}(\texttt{rec}(1, \kappa_2^S(i, n)), x, \pi_1^S(x)))] :: map(\beta(1))(equal'4)
\end{array}$$

Then, projection is specified as below.

$$\begin{array}{l}
\texttt{LET}\ i = \pi_1^S(\texttt{vr}),\ n = \pi_1^S(\pi_2^S(\texttt{vr})),\ x = \pi_2^S(\pi_2^S(\texttt{vr}))\ \texttt{IN} \\
k_4(a, b, c, d) = \kappa_2^S(a, \kappa_2^S(b, \kappa_2^S(c, d))) \\
proj'4 := \\
[\texttt{rec}(1, k_4(\texttt{cnst}(0), i, n, x))] :: map(\beta(1))(proj_aux'4)
\end{array}$$

The correctness of the projection is formalized by next lemma.

**Lemma 4 (Correctness of Projection)**

$$\begin{array}{l}
\forall(i, x) : \\
\ \forall(n \mid i \leq n) : \\
\quad \gamma(proj)(\kappa_2(i, \kappa_2(n, x)), nat2list(n, x)(i))
\end{array}$$

The analysis of composition requires the functions *exprComp* and *chainOffset* below. In these functions, $l$ is a non-empty list of list of expressions that is the kernel of a `PVS0` program to be composed. The idea is simulate a 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. But 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.

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

$$
\begin{array}{l}
\mathit{chainOffset}(n, l) := \\
\ \texttt{IF}\ \ |l| = 1\ \texttt{THEN}\ \mathit{map}(\beta(n))(l(0)); \\
\ \texttt{ELSE}\ \ \mathit{map}(\beta(n))(l(0)) :: \mathit{chainOffset}(n + |l(0)|, \mathit{cdr}(l));
\end{array}
$$

Let $F$ be a `PVS0` program, $L$ a non-empty list of `PVS0` programs, $f := F'4$ and $l := \mathit{map}(\lambda(a, b, c, d) : d)(L)$. 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* 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* 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 $F$.

$$
\begin{array}{rl}
\mathit{comp}(f, l)'4 := & [\texttt{rec}(1, \kappa_2^S(\texttt{cnst}(|l|), \\
& \ \mathit{exprComp}(1 + |f|, l)))] :: \\
& \mathit{chainOffset}(1, [f] :: l))
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{list2nat}(l) := \\
\ \texttt{IF}\ \ |l| = 1\ \texttt{THEN}\ l(0); \\
\ \texttt{ELSE}\ \ \kappa_2(l(0), \mathit{list2nat}(\mathit{cdr}(l)));
\end{array}
$$

Now, it is possible to establish the composition lemma as follows.

**Lemma 5 (Correctness of Composition)**

$$
\begin{array}{l}
\texttt{LET}\ O_1 = [\mathit{succ}, \pi_1, \pi_2],\ O_2 = [\mathit{greater}, \kappa_2]\ \texttt{IN} \\
\forall(f, l|\ |l| > 0) : \forall(v_i, v_o) : \\
\ \gamma(\mathit{comp}(f, l))(v_i, v_o) \Leftrightarrow \\
\ \exists(l_n|\ |l_n| = |l|) : \\
\ \ \gamma(O_1, O_2, 0, f)(\kappa_2(|l_n|, \mathit{list2nat}(l_n)), v_o) \wedge \\
\ \ \forall(i|\ i < |l_n|) : \gamma(O_1, O_2, 0, l(i))(v_i, l_n(i))
\end{array}
$$

Minimization of a `PartialRecursiveTC PVS0` Program $(O_1, O_2, 0, f)$ uses the function *min_aux* specified below.

$$
\begin{array}{l}
\mathit{min\_aux}(f)'4 := \\
\ [\texttt{ite}(\texttt{rec}(1, \texttt{vr}), \\
\qquad \texttt{rec}(0, \kappa_2^S(\mathit{succ}^S(\pi_1^S(\texttt{vr})), \pi_2^S(\texttt{vr}))), \\
\qquad \pi_1^S(\texttt{vr}))] :: \\
\ \mathit{map}(\beta(1))(f)
\end{array}
$$

Minimization $(O_1, O_2, 0, f)$ of is specified below:

$$
\begin{array}{l}
\mathit{min}(f)'4 := \\
\ [\texttt{rec}(1, \kappa_2^S(\texttt{cnst}(0), \texttt{vr}))] :: \\
\ \mathit{map}(\beta(1))(\mathit{min\_aux}(f)'4)
\end{array}
$$

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

**Lemma 6 (Correctness of Minimization)**

$$
\begin{array}{l}
\texttt{LET } O_1 = [succ, \pi_1, \pi_2],\ O_2 = [greater, \kappa_2] \texttt{ IN} \\
\ \forall(y, f, ans) : \\
\ \ \gamma(min(f))(y, ans) \Leftrightarrow \\
\ \ (\gamma(O_1, O_2, 0, f)(\kappa_2(ans, y), 0) \wedge \\
\ \ \forall(i | i < ans) : \\
\ \ \ \exists(k | k > 0) : \gamma(O_1, O_2, 0, f)(\kappa_2(i, y), k))
\end{array}
$$

In order to show the result for primitive recurrence, the subtraction of a tuple of naturals codified as a unique natural is specified as:

$$
\begin{array}{l}
\texttt{LET } x = \pi_1^S(\texttt{vr}),\ y = \pi_2^S(\texttt{vr}) \texttt{ IN} \\
sub'4 := \\
\ [\texttt{ite}(greater^S(x, y), \\
\qquad succ^S(\texttt{rec}(0, \kappa_2^S(x, succ^S(y)))), \\
\qquad \texttt{cnst}(0))]
\end{array}
$$

Using *sub*, the subtraction by 1 is specified:

$$
\begin{array}{l}
sub1'4 := \\
\ [\texttt{rec}(1, \kappa_2^S(\texttt{vr}, \texttt{cnst}(1)))] :: map(\beta(1))(sub'4)
\end{array}
$$

The primitive recurrence is given by the `PVS0` program:

$$
\begin{array}{l}
\texttt{LET } x = \pi_1^S(\texttt{vr}), \\
\qquad y = \pi_2^S(\texttt{vr}), \\
\qquad less_1(e) = \texttt{rec}(1 + |recur| + |final|, e), \\
\qquad recur\_fun(e_1, e_2, e_3) = \\
\qquad\ \texttt{rec}(1, \kappa_2^S(e_1, \kappa_2^S(e_2, e_3))), \\
\qquad final\_fun(e) = \texttt{rec}(1 + |recur|, e), \\
\qquad recursive\_call(e_1, e_2) = \texttt{rec}(0, \kappa_2^S(e_1, e_2)) \\
\texttt{IN} \\
prim\_recurrence(recur, final)'4 := \\
\ [\texttt{ite}(x, \\
\qquad recur\_fun(recursive\_call(less_1(x), y), less_1(x), y), \\
\qquad final\_fun(y))] :: \\
\ map(\beta(1))(recur) :: map(\beta(1 + |recur|))(final) :: \\
\ map(1 + |recur| + |final|)(sub1'4)
\end{array}
$$

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

**Lemma 7 (Primitive Recurrence)**

$$
\begin{array}{l}
\forall(recur, final) : \forall(x, y, ans) : \\
\ \gamma(prim\_recurrence(recur, final))(\kappa_2(x, y), ans) \ \Leftrightarrow \\
\ \exists(l \mid x + 1 = |l|) : ans = l(|l| - 1) \wedge \\
\ \gamma(O_1, O_2, 0, final)(y, l(0)) \wedge \\
\ \forall(i \mid i < |l| - 1) : \\
\ \gamma(O_1, O_2, 0, recur)(\kappa_2(l(i), \kappa_2(i, y)), l(i + 1))
\end{array}
$$

Correctness of the composition requires some extra lemmas. For example, showing that *comp* generates a `PVS0` program of the type `PartialRecursiveTC` requires to show that the indices of the symbol `rec` generated by the functions *exprComp* and *chainOffset* are valid. In addition, an induction on the length of $l$ solves the goal.

On the other hand, correctness of minimization and primitive recursion uses extras inductive predicates.

$$
\begin{array}{l}
min\_relation(x, y, f, ans) := \\
\ \texttt{IF}\ \ \gamma(O_1, O_2, 0, f)(\kappa_2(x, y), 0)\ \texttt{THEN}\ ans = x; \\
\ \texttt{ELSIF}\ \ \exists(k) : \gamma(O_1, O_2, 0, f)(\kappa_2(x, y), k) \\
\ \texttt{THEN}\ \ min\_relation(x + 1, y, f, ans); \\
\ \texttt{ELSE}\ \ False.
\end{array}
$$

$$
\begin{array}{l}
prim\_recurrence\_relation(recur, final)(x, y)(ans) := \\
\texttt{IF}\ \ x \neq 0\ \texttt{THEN}\ \ \exists(z) : \\
\ \gamma(O_1, O_2, 0, recur)(\kappa_2(z, \kappa_2(x - 1, y), ans)) \wedge \\
\ prim\_recurrence\_relation(recur, final)(x - 1, y)(ans); \\
\texttt{ELSE}\ \ \gamma(O_1, O_2, 0, final)(y, ans)).
\end{array}
$$

Using these predicates instead of using the notions of minimization and primitive recurrence in the formalizations has as advantage that it avoids exhaustive expansions of the definition the $\varepsilon$ predicate. The evaluation predicate $\varepsilon$ contains several existential quantifiers that require being Skolemized and in the most complex case be instantiated. On the other side, each inductive predicate specified in PVS has an associated inductive schema associated, which simplifies the formal proofs.

For proving the correctness of the minimization, first, the sufficiency was formalized using the inductive schema given by the predicate *min_relation*. Then, the necessity was formalized using the equivalence between the function $\chi$ and the predicate $\varepsilon$. As aforementioned, the function $\chi$ provides the measure to be applied in inductive proofs. This kind of inductive proof is just the one performed for formalizing the necessity. In addition, the primitive recurrence is formalized with a simple induction on $x$.

## 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 own 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 there exists the possibility of creating a partial recursive `PVS0` program from a list of `PVS0` expressions such that its output for any evaluation is itself, then Rice's Theorem holds.

The formalization uses the same basic operators for successor, projection, greater-than and the bijection $\kappa_2$ used to prove Turing Completeness for the `PVS0` model. The result is specified as below.

**Theorem 1 (Recursion Theorem)**

$$\begin{aligned}
&\forall(E_f) : \exists(print : \texttt{PartialRecursive}) : \\
&\quad \texttt{LET} \quad self = (\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, E_f :: map(\beta(|E_f|))(print'4)) \ \texttt{IN} \\
&\qquad\qquad partial_recursive_TC(self) \ \wedge \\
&\qquad\quad \forall(i) \ : \varepsilon(self)(\beta(|E_f|)(print'4(0)), i, \kappa_p(self))
\end{aligned}$$

*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 behaviour of *self* that is to be able to calculate its own Gödel number and 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 $[\texttt{cnst}(\alpha_{aux}(map(\kappa_e(|E_f :: A|))(E_f :: A)))]$, such that $self'4 = E_f :: A :: [\texttt{cnst}(\alpha_{aux}(map(\kappa_e(|E_f :: A|))(E_f :: A)))]$. 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 $\alpha_{aux}$ was recursively specified from the back to front to be adapted to *self'*4 in which a number related to the first element is calculated before that another number related to the last element is calculated. This decision reduces the effort necessary in the formalization avoiding analysis of a specification in which the last element of an alternative version of *self'*4 should represent a stack of naturals associated to each element in $E_f :: A$ by the function $\kappa_e$. In this case, to calculate the the Gödel number of this alternative version of *self* it would be necessary to add the number associated to its last element to the bottom of the stack.

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

$$A := \beta(|E_f|)(printA(\delta, |E_f|))$$

The function *printA* is specified below.

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

Above $[\texttt{vr}]^{len}$ is a list with *len* repetitions of `vr`. The list for *mult* is specified to receive a natural input, apply the bijective function $\kappa_2^{-1}$ to obtain a pair of naturals and multiplying them, as below.

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

Since in the specification of $A$ above the arguments of *printA* are $\delta$ 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 codified as a unique natural by the function $\kappa_2$ as below.

$$
\begin{array}{l}
sum := [\mathtt{ite}(\pi_1^S(\mathtt{vr}), \\
\qquad\qquad succ(\mathtt{rec}(0, \kappa_2^S(\mathtt{rec}(1, \pi_1^S(\mathtt{vr})), \pi_2^S(\mathtt{vr})))), \\
\qquad\qquad \pi_2^S(\mathtt{vr}))] \\
:: \beta(1)(sub1'4)
\end{array}
$$

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.

$$sum_f(x, y) = \mathtt{IF}\ x \neq 0\ \mathtt{THEN}\ 1 + sum_f(x - 1, y)\ \mathtt{ELSE}\ y$$

$$mult_f(x, y) = \mathtt{IF}\ x \neq 0\ \mathtt{THEN}\ y + mult_f(x - 1, y)\ \mathtt{ELSE}\ 0$$

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` specificed 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*)**

$$
\begin{array}{l}
\forall(i, len, len2, h) : \\
\quad \gamma(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, printA(len, len2) :: [\mathtt{cnst}(h)]) \\
\qquad (i, \kappa_2(1 + len + len2 + |mult|, \kappa_2(h, 5 \times h + 1)))
\end{array}
$$

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

$$
\begin{array}{l}
\forall(i) : \\
\quad \gamma(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, printA(\delta, |E_f|) :: [\mathtt{cnst}(h)]) \\
\qquad (i, \kappa_2(|self'4| - 1, \kappa_2(h, \kappa_e(|self'4| - 1)(\mathtt{cnst}(h)))))
\end{array}
$$

because,

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

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

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

The result of this replacement is:

$$\begin{array}{l}\forall(i) : \\ \quad \gamma(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, printA(\delta, |E_f|) :: [\texttt{cnst}(h)]) \\ \qquad (i, \kappa_2(|self'4| - 1, \alpha_{aux}(map(\kappa_e(|self'4| - 1))(self'4))))\end{array}$$

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

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

Thus, it can be concluded that:

$$\begin{array}{l}\forall(i) : \\ \gamma(\boldsymbol{O_1}, \boldsymbol{O_1}, \bot, printA(\delta, |E_f|) :: [\texttt{cnst}(h)])(i, \kappa_p(self))\end{array}$$

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

The most difficult part of this formalization is related with the construction of the specialized Gödelization functions required to build *self*. One challenge in the implementation of $\kappa_p$ was to build it in such manner that it facilitates further steps of the formalization. An appropriate function $\alpha_{aux}$ was enough to reach this aim. Specifically for the Recursion Theorem it is not required $\kappa_p$ to be bijective, but 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 for the Fixed-Point Theorem. Ensuring bijectivity of $\kappa_p$ was technically difficult since it required every necessary auxiliary function to be bijective as well. For some auxiliary components the formalization was straightforward, but for another 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 arise when specific kernels of `PVS0` programs were considered such as those that included only valid recursive call indices. An example of such lemmas on lists that fails is $|A :: B| = |A| + |B|$ for which there is an appropriate lemma ready for it, but if the types of $A$ and $B$ are a subtype of $A :: B$, also input of the length function ($|\_|$), the lemma needs to be specialized and proved separately. The general, non-provided in PVS, solution is to show $|A|[T] = |A|[S]$, where $S$ is subtype of $T$. Several similar type inference problems were solved when formalizing the Recursion Theorem (in theories that were not includeded in Figure 1).

## 6 Rice's Theorem

The formalization of the Rice's Theorem is a corollary of the Recursion Theorem. It is proved that if the basic built-in operators imply that the Recursion Theorem holds then the Rice's Theorem for this theory also holds. Notice, that the basic operators used in the theory `mf_pvs0_Recursion_Theorem` to guarantee this theorem, and those used in theory `mf_pvs0_Turing_Completeness` to guarantee Turing Completeness are the same. The formalization in this section proves that for all Göedelizations that the Recursion Theorem holds the Rice's Theorem also holds. Furthermore, it follows Cantor's diagonal argument, similarly to standard proofs of the undecidability of the Halting Problem and uncountability of real numbers. An alternative formalization approach of Rice's Theorem could have been based on the construction of a universal program for the `PVS0` model, but it would increase the complexity of the formalization. Using such construction, the proof would require reducing the Halting Problem to the problem of separability of semantic properties of `PVS0` programs, which is not the case of the current formalization. Thus, the formalization of the Rice's Theorem depends only on the above mentioned Theorem 1, and does not require undecidability of the Halting Problem.

### 6.1 Formalization of Rice's Theorem

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

**Theorem 2 (Rice's Theorem)**

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

*Proof* **Necessity**: Suppose that $P = \mathit{fullset}$. Let $\top$ be an element different from $\bot$. The `PVS0` program $\mathit{decider} = (\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(\top)])$, decides $\mathit{fullset}$. Now, suppose that $P = \emptyset$. The `PVS0` program $\mathit{decider} = (\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(\bot)])$ decides $\emptyset$.

**Sufficiency**: Proved by contraposition. Let assume that $(P \neq \mathit{fullset} \wedge P \neq \emptyset)$. This implies that there are `PVS0` programs, say $p$ and $np$, such that $P(p)$ and $\neg P(np)$. For reaching a contradiction, suppose that there exists $\mathit{decider} : \texttt{Computable}$ such that:

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

And, consider the program *opp* with kernel:

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

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

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

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

$$\begin{array}{l}self(n) \;:= \\ \quad \texttt{IF} \;\; decider(\kappa_p(self)) \neq \bot \;\texttt{THEN}\; np(n); \;\texttt{ELSE}\; p(n);\end{array}$$

The proof uses Cantor's diagonal argument. If $decider(\kappa_p(self)) \neq \bot$, then $P(self)$, but *self* behaves as $np$ and thus $\neg P(self)$ holds, which is a contradiction. Otherwise, if $decider(\kappa_p(self)) = \bot$, 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 aforementioned theorem 1 implies that there exists an element of the partial recursive class, say *print*, such that:

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

Making $pvso = self$ it can be concluded that

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

The proof splits into two sub-cases.

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

Since $P$ is a semantic predicate, one has:

$$\begin{array}{c}\forall(pvso_1, pvso_2) : \\ (\forall(i, o) : \gamma(pvso_1)(i, o) \Leftrightarrow \gamma(pvso_2)(i, o)) \Rightarrow \\ (P(pvso_1) \Leftrightarrow P(pvso_2))\end{array}$$

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

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

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

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

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

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

$$\varepsilon(self)(opp :: map(\beta(|opp|))(print`4)(0), i, o)$$

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

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

Then, by the definition of $\varepsilon$ and operational semantics of `ite`, one has:

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

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

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

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

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

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

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

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

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

By using the shift code lemma (Lemma 1),

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

Thus one obtains,

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

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

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

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

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

Applying the shift code lemma (Lemma 2):

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

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

**Sub-case 2**: $\neg P(self)$. It follows analogously to sub-case 1, except for the supposition that $P$ is a semantic predicate where $pvso_1$ and $pvso_2$ are instantiated respectively as *self* and $p$, which leads to a contradiction.

### 6.2 Applications of Rice's Theorem

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

**Corollary 1 (Undecidability of the Uniform Halting Problem)**

$$\neg\exists(decider : \texttt{Computable}) : \\ \forall(pvso : \texttt{PartialRecursive}) : \\ \neg(\gamma(decider)(\kappa_p(pvso), \bot) \Leftrightarrow T_\varepsilon(pvso))$$

*Proof* The formalization uses Rice's Theorem instantiating the semantic predicate as $T_\varepsilon$. The predicate $T_\varepsilon$ is a semantic predicate because if two `PVS0` programs perform the same, both are either terminating or not. Since the set $T_\varepsilon$ is neither equal to the empty set nor to the whole set `PartialRecursive`, there exists no computable decider for this set. To prove this, it is shown that the `PartialRecursive` constant program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(0)])$ belongs to $T_\varepsilon$, while a simple loop `PartialRecursive` program specified as $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{rec}(0, \texttt{vr})])$ does not.

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

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

**Corollary 2 (Undecidability of Existence of Fixed Points)**

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

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

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

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 `PartialRecursive`. Indeed, on one side, the predicate holds for the program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(0)])$, showing that it is different from empty set. On the other side, it does not hold for the program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{op2}(i, \texttt{vr}, \texttt{cnst}(1))])$ that performs the same as $\lambda(n : \mathbb{N}) : \kappa_2(n, 1)$, concluding that the predicate is not equal to `PartialRecursive`.

**Corollary 3 (Undecidability of Self Replication)**

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

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

$$\begin{array}{l}\lambda(pvso : \texttt{PartialRecursive}) : \\ \quad \exists(p : \texttt{PartialRecursive}) : \\ \quad\quad \forall(i) : \gamma(p)(i, \kappa_p(p)) \wedge \gamma(pvso)(i, \kappa_p(p))\end{array}$$

The predicate above is a semantic predicate because if two `PVS0` programs perform the same, either both returns 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 `PartialRecursive` set. Using the assumption of the Recursion Theorem and instantiating it with $[\texttt{rec}(1, \texttt{vr})]$, one shows that the predicate is not empty. On the other side, the program $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{op2}(i, \texttt{cnst}(1), \texttt{vr})])$ shows that the predicate is not the whole `PartialRecursive` set.

**Corollary 4 (Undecidability of Functional Equivalence)**

$$\begin{array}{l}\neg\exists(decider : \texttt{Computable}) : \\ \quad \forall(pvso_0, pvso_1 : \texttt{PartialRecursive}) : \\ \quad\quad (\neg\gamma(decider)(\kappa_2(\kappa_p(pvso_0), \kappa_p(pvso_1)), \bot) \Leftrightarrow \\ \quad\quad\quad (\ \forall(v_i, v_o) : \gamma(pvso_0)(v_i, v_o) \Leftrightarrow \gamma(pvso_1)(v_i, v_o)\ )\quad )\end{array}$$

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

$$\begin{array}{l}\lambda(pvso : \texttt{PartialRecursive}) : \\ \quad \forall(v_i, v_o) : \\ \quad\quad \gamma(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(0)])(v_i, v_o) \Leftrightarrow \gamma(pvso)(v_i, v_o)\end{array}$$

It is a semantic predicate because either two `PVS0` programs always returns zero or not.

To show that the predicate is neither the empty not the full `PartialRecursive` set, it is enough to prove that the constant zero program belongs to the predicate and that the constant one program does not. After that, one uses the assumed program *decider* to build another program for the equivalence with the constant zero program; this program is built as $(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{rec}(1, \texttt{op2}(i, \kappa_p(\boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\texttt{cnst}(0)]), vr))] :: decider`‘4), where $i$ is the index to the $\kappa_2$ function. Using the shifting code lemmas, it can indeed be simplified to deciding equivalence to the constant zero program.

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

## 7 Another Formalized results and Related Work

### 7.1 Another Formalized results

As mentioned in the introduction, the development also includes proofs of other results, such as the Undecidability of the Halting Problem. This theorem was formalized following the classical diagonalization proof method. To obtain a contradiction, 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 specific input. The input of *oracle* is a natural that codifies a pair of a `PVS0` program and a natural input. The codification uses an arbitrary Gödelization in order 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 infinity loop, otherwise, it returns the codified 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.

The specification in PVS of the undecidability of the Halting Problem for the multiple-function `PVS0` model is given in the theorem below.

**Theorem 3 (Undecidability of the Halting Problem for `PVS0`)** *For all $O_1$, $O_2$, $\bot$, and $\kappa_p$, there is no program oracle of type* `Computable` *such that for all*

*pvso* $= (O_1, O_2, \bot, E_f)$ *of type* `PartialRecursive` *and for all* $n \in \mathbb{N}$,

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

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

Another interesting result formalized in this development is The Fixed-Point Theorem. It states that for any program $f$ that transforms a program in 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 argument, and split it into two another 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`)** *For all $f$ of type* `Computable`*, there exists $p$ of type* `PartialRecursive`*, 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, $\Delta$ 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 $\Delta(diagonal)(f \circ diagonal)$, where *diagonal* is as described in the previous paragraph.

### 7.2 Related work

Nowadays, mechanical proofs of computability properties is not only of interest as an exercise of formalization, but also of great importance to provide formal support for computational models used for pragmatical issues. As mentioned in the introduction, the main aim of the computational 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 not only consist of a single function, but also were constrained in inductive levels such that in the level zero only the basic functions successor, greater-than and projections are allowed and, in subsequent levels, other `Computable` functions can be specified allowing

calls to functions of the previous level as operators. Building composition of such PVS0 programs was not straightforward, which makes 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 application of the operator $\beta$.

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

The equivalence between termination criteria was formalized for the single-function PVS0 model considered in [7]. However, such an equivalence has not been formalized for the current multiple-function PVS0 model. Theoretically, all termination criteria mentioned in the introduction (references [19], [20], [4], [27], [3]) work for both models, but technically, some of the criteria require a re-adaptation to deal simultaneously with static analysis of multiple-function programs that allow even mutual recursion (which, in particular, is avoided in the PVS functional specification language).

Computability properties have been formalized since the development of the first theorem provers and proof assistants. As well-known examples one can mention the mechanical proof of the undecidability of the Halting Problem in [5] using the LISP language as model of computation, as well as the formalization in Agda of the same theorem in [15] using as a model of computation 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], Foster and Smolka used as model of computation call-by-value lambda calculus, which is a Turing Complete model of computation, where beta-reduction can be applied 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 including Rice's Theorem; indeed, they formalized that semantic predicates such that there are elements both in them and in their complements, are not recognizable. This property is called Rice's Lemma by the authors and it is used to conclude the Rice's Theorem. Also, Norrish formalized in [21], 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. The Rice's Theorem was also formalized by Carneiro in Lean [6]. This formalization uses partial recursive functions as model of computation, and the proof uses the Fixed-Point Theorem to conclude the Rice's Theorem as a corollary; also, in this work the undecidability of the Uniform Halting Problem is obtained as a corollary.

As previously discussed, the model of computation chosen influences both the level of difficulty of the formalizations of computability results, and the manner in which undecidability results are proved by reduction from other results selected as starting point of such formalizations. The unique particular such choice in or

formalization refers to the reduction from the Recursion to the Rice's Theorem. Although there are well-known properties followed by textbooks' proofs, such as assumption of existence of a universal machine or undecidability of the universal language (e.g., [23], [14]), 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 simplest manner to formalize the Rice's Theorem was as a corollary of the Recursion Theorem using as basic built-in operators $\kappa_2$, successor, the projections composed to $\kappa_2^{-1}$ and greater-than, but it should be noticed 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 interesting computability results formalized over linguistic computational models. Forster, Kirsk and Smolka formalized in [10] undecidability of validity, satisfiability, and provability of first-order formulas following a synthetic approach based on the computation native to Coq's constructive type theory. Forster and Larchey-Wendling formalized in [11] using Coq, the reduction of the Post Correspondence Problem (PCP) via binary stack machines and Minsky machines to provability of intutionistic 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 intutionistic linear logic. Also, Larchey-Wendling formalized in [17] that any function of the type $\mathbb{N}^k \to \mathbb{N}$ specified using Coq is total. Recently, previous author together with Forest formalized in [18] 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 of Diophantine equations. More recently, Spies and Forster [24], and Kirst and Larchey-Wendling [16] added two interesting formalizations to the Coq library of synthetic undecidability proofs, namely, formalizations of the undecidability of higher order unification and undecidability of first-order satisfiability by finite models (FSAT). The former result formalizes Goldfarb's proof of 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 the Trakhtenbrot's Theorem, originally proved by a reduction from the Halting Problem for Turing Machines [25]. The formalization in [16] is obtained by a reduction from PCP and includes a sharper version of 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 seen as a model of computation. Turing completeness of `PVS0` was formalized in 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 build tuples from naturals and vice versa. The proof consists in formalizations of correctness of `PVS0`

implementations of these functions and of operators for composition, primitive recurrence and minimization.

Additionally, a formalization of Rice's Theorem is given for PVS0 was given. The formalization uses the Recursion Theorem (also formalized in PVS) 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 the Rice's Theorem includes formalizations of corollaries such as undecidability of the uniform Halting Problem, functional equivalence problem, 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 by PVS. The quantitative data of the files of proofs in Figure 1, is given in Table 3. Data of other auxiliary theories that required a few 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 |
|---|---|---|---|
| mf_pvs0_Rices_Theorem | 5206 loc - 594K | 2 | 2 |
| mf_pvs0_Recursion_Theorem | 6754 loc - 572K | 7 | 14 |
| mf_pvs0_Turing_Completeness | 17677 loc - 1,5M | 27 | 32 |
| mf_pvs0_Fixedpoint | 4712 loc - 68K | 1 | 4 |
| mf_pvs0_Halting | 1704 loc - 149K | 2 | 3 |
| mf_pvs0_Rices_Theorem_Corollaries | 1786 loc - 100K | 5 | 0 |
| mf_pvs0_basic_programs | 4879 loc - 319K | 9 | 10 |
| mf_pvs0_computable | 6586 loc - 310K | 9 | 50 |
| mf_pvs0_lang | 2817 loc - 122K | 20 | 13 |
| mf_pvs0_expr | 3418 loc - 166K | 7 | 47 |

Despite the fact that 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 with technical and simple issues for which semantic evaluation is required (i.e., expansions of the definition of the predicate $\varepsilon$) and simple instantiations of premises existentially quantified. In addition, 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 with 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 proper-

ties of computational models such as the Word Problem for algebraic structures ([22]) and Hilbert's Tenth Problem [18].

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 in order to deal with undecidability of problems outside of the field of computability such as the Word Problem over algebraic structures and, SAT (as done in [16] 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. A. Alves Almeida and M. Ayala-Rincon. Formalizing the Dependency Pair Criterion for Innermost Termination. *Science of Computer Programming*, 195(102474), 2020.
2. T. Arts. Termination by Absence of Infinite Chains of Dependency Pairs. In *21st International Colloquium on Trees in Algebra and Programming CAAP*, volume 1059 of *LNCS*, pages 196–210. Springer, 1996.
3. T. Arts and J. Giesl. Termination of term rewriting using Dependency Pairs. *Theoretical Computer Science*, 236:133–178, 2000.
4. A. B. Avelar. *Formalização da Automação da Terminação Através de Grafos com Matrizes de Medida*. PhD thesis, Department of Mathematics, Universidade de Brasília, 2014. In Portuguese.
5. R. S. Boyer and J. S. Moore. A Mechanical Proof of the Unsolvability of the Halting Problem. *Journal of the Association for Computing Machinery*, 31(3):441–458, 1984.
6. M. Carneiro. Formalizing Computability Theory via Partial Recursive Functions. In *10th International Conference on Interactive Theorem Proving ITP*, volume 141 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
7. T. M. Ferreira Ramos, C. A. Muñoz, M. Ayala-Rincón, M. M. Moscato, A. Dutle, and A. Narkawicz. Formalization of the Undecidability of the Halting Problem for a Functional Language. In *25th International Workshop on Logic, Language, Information, and Computation WoLLIC*, volume 10944 of *LNCS*, pages 196–209. Springer, 2018.
8. R. W. Floyd and R. Beigel. *The Language of Machines: An Introduction to Computability and Formal Languages*. W H Freeman & Co, 1994.
9. Y. Forster, E. Heiter, and G. Smolka. Verification of PCP-Related Computational Reductions in Coq. In *9th International Conference on Interactive Theorem Proving ITP*, volume 10895 of *LNCS*, pages 253–269. Springer, 2018.
10. Y. Forster, D. Kirst, and G. Smolka. On Synthetic Undecidability in Coq, with an Application to the Entscheidungsproblem. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs CPP*, pages 38–51. ACM, 2019.
11. Y. Forster and D. Larchey-Wendling. 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*, pages 104–117. ACM, 2019.
12. Y. Forster and G. Smolka. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *8th International Conference on Interactive Theorem Proving ITP*, volume 10499 of *LNCS*, pages 189–206. Springer, 2017.
13. W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.
14. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, third edition, 2008.

15. K. Johannisson. Formalizing the halting problem in a constructive type theory. In *International Workshop on Types for Proofs and Programs TYPES*, volume 2277 of *LNCS*, pages 145–159. Springer, 2000.
16. D. Kirst and D. Larchey-Wendling. Trakhtenbrot's Theorem in Coq, A Constructive Approach to Finite Model Theory. *CoRR*, abs/2004.07390, 2020.
17. D. Larchey-Wendling. Typing Total Recursive Functions in Coq. In *8th International Conference on Interactive Theorem Proving ITP*, volume 10499 of *LNCS*, pages 371–388. Springer, 2017.
18. D. Larchey-Wendling and Y. Forster. Hilbert's Tenth Problem in Coq. In *4th International Conference on Formal Structures for Computation and Deduction FSCD*, volume 131 of *LIPIcs*, pages 27:1–27:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.
19. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The Size-Change Principle for Program Termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2001.
20. P. Manolios and D. Vroon. Termination Analysis with Calling Context Graphs. In *18th International Conference on Computer Aided Verification CAV*, volume 4144 of *LNCS*, pages 401–414. Springer, 2006.
21. M. Norrish. Mechanised Computability Theory. In *Second International Conference on Interactive Theorem Proving ITP*, volume 6898 of *LNCS*, pages 297–311. Springer, 2011.
22. E. L. Post. Recursive unsolvability of a problem of Thue. *The Journal of Symbolic Logic*, 12(1):1–11, 1947.
23. M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, third edition, 2012.
24. S. Spies and Y. Forster. Undecidability of higher-order unification formalised in Coq. In *9th ACM SIGPLAN International Conference on Certified Programs and Proofs CPP*, pages 143–157. ACM, 2020.
25. B. A. Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. *Doklady Akademii Nauk SSSR*, 70(4):569–572, 1950.
26. A. M. Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937.
27. A. M. Turing. Checking a Large Routine. In M. Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.