Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Verifying the Computational Properties of a First-Order Functional Model

(Verificação das Propriedades Computacionais de um Modelo Funcional de Primeira-Ordem)

Thiago Mendonça Ferreira Ramos

Document submitted in partial fullfilment of
the requirements to Doctoral Degree in Informatics

Advisor
Dr. Mauricio Ayala-Rincón

Co-advisor
Dr. César Augusto Muñoz

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Verifying the Computational Properties of a First-Order Functional Model

(Verificação das Propriedades Computacionais de um Modelo Funcional de Primeira-Ordem)

Thiago Mendonça Ferreira Ramos

Document submitted in partial fullfilment of
the requirements to Doctoral Degree in Informatics

Brasília, June 15, 2023

Dedicatória

Dedico essa tese a Nossa Senhora, soberana do meu coração. Dedico também a minha mãe, Ana Lucia Mendonça Ferreira Ramos, ao meu pai, Edson Sergio Ferreira Ramos e a minha irmã, Mariana Mendonça Ferreira Ramos. Por fim, dedico essa tese a minha futura esposa, a quem eu não conheço, ou, quem sabe, eu já a conheça.

I dedicate this thesis to Our Lady, sovereign of my heart. I also dedicate it to my mother, Ana Lucia Mendonça Ferreira Ramos, my father, Edson Sergio Ferreira Ramos, and my sister, Mariana Mendonça Ferreira Ramos. Finally, I dedicate this thesis to my future wife, whom I do not know, or, perhaps, I already know her.

Agradecimentos

Agradeço à Santíssima Trindade pela oportunidade de terminar o doutorado. Agradeço ao meu orientador Mauricio Ayala-Rincón e ao meu co-orientador, César Augusto Muñoz pelas valiosas correções e sugestões. Agradeço a meu santo de devoção, São Padre Pio e ao meu anjo da guarda pelas inspirações ao longo dessa jornada. Agradeço a Nosso Senhora do Rosário de Fátima. Agradeço ao Beato Carlo Acutis, que se fosse vivo, teriamos a mesma idade e que por seu talento em informática também foi fonte de inspirações. Agradeço aos meus amigos: Ariane Alves Almeida, Gabriel Ferreira Silva, Santiago Miler Quispe Mamani, Pavel Sejas Paz, Geovane Cardoso de Brito, Ana Cristina Valverde, Daniel Saad Nunes, Lucas Ângelo da Silveira, Washington Ribeiro Segundo.

I thank the Holy Trinity for the opportunity to finish my PhD. I thank my advisor Mauricio Ayala-Rincón and my co-supervisor, César Augusto Muñoz for their valuable corrections and suggestions. I thank my saint of devotion, St. Padre Pio, and my guardian angel for the inspiration throughout this journey. I thank Our Lady of the Rosary of Fatima. I thank Blessed Carlo Acutis, who if he were alive, we would be the same age, and who for his talent in computing was also a source of inspiration. I thank my friends: Ariane Alves Almeida, Gabriel Ferreira Silva, Santiago Miler Quispe Mamani, Pavel Sejas Paz, Geovane Cardoso de Brito, Ana Cristina Valverde, Daniel Saad Nunes, Lucas Ângelo da Silveira, Washington Ribeiro Segundo.

Resumo Expandido

Este trabalho descreve a mecanização de propriedades computacionais de um modelo funcional que tem sido aplicado para automatizar o raciocínio sobre a terminação de programas. A formalização foi desenvolvida no assistente de provas de lógica de ordem superior, chamado *Prototype Verification System* (PVS). O modelo de linguagem foi projetado para imitar o fragmento de primeira ordem de especificações funcionais e é chamado `PVS0`. Foram considerados dois modelos computacionais: o primeiro modelo especifica um programa funcional por meio de uma única função (modelo *single-function* `PVS0`, ou `SF-PVS0`), e o segundo modelo permite a especificação simultânea de múltiplas funções (modelo *multiple-function* `PVS0`, ou `MF-PVS0`). A semântica operacional da recursão na especificação do modelo `SF-PVS0`suporta a recursão sobre o programa completo. Por outro lado, em programas `MF-PVS0`, as chamadas funcionais são permitidas para todas as funções especificadas no programa. Este trabalho tem como objetivo certificar matematicamente a robustez dos modelos `PVS0` como modelos computacionais universais. Para isso, propriedades cruciais e teoremas foram formalizados, incluindo Turing Completude, a indecidibilidade do Problema da Parada, o teorema da recursão, o teorema de Rice e o teorema do Ponto Fixo. Além disso, o trabalho discute avanços na indecidibilidade do Problema da Palavra e do Problema da Correspondência de Post. A indecidibilidade do Problema da Parada foi formalizada considerando a avaliação semântica de programas `PVS0` que foram aplicados na verificação da terminação de especificações em PVS. A equivalência entre a avaliação funcional e predicativa de operadores foi fundamental para esse objetivo. Além disso, a composicionalidade de programas `MF-PVS0`, habilitada diretamente pela possibilidade de chamar diferentes funções, torna fácil a formalização da Turing Completude. Portanto, enriquecer o modelo foi uma decisão de design importante para simplificar a mecanização dessa propriedade e dos teoremas mencionados acima.

Palavras-chave: Turing Completude, Problema da Parada, Teorema de Rice, Teorema do Ponto Fixo, Problema da Correspondencia de Post, Problema da Palavra, Indecidibilidade

Abstract

This work describes the mechanization of the computational properties of a functional-language model that has been applied to reasoning about the automation of program termination. The formalization was developed using the higher-order proof assistant Prototype Verification System (PVS). The language model was designed to mimic the first-order fragment of PVS functional specifications and is called `PVS0`. Two different computational models are considered: the first model specifies functional programs through a unique function (single-function `PVS0` model, or `SF-PVS0`), and the second model allows simultaneous specification of multiple functions (multiple-function `PVS0` model, or `MF-PVS0`). This work aims to mathematically certify the robustness of the `PVS0` models as universal computational models. For doing that, crucial properties and theorems were formalized, including Turing Completeness, the undecidability of the Halting Problem, the Recursion Theorem, Rice's Theorem, and the Fixed Point Theorem. Furthermore, the work discusses advances in the undecidability of the Word Problem and the Post Correspondence Problem.

The undecidability of the Halting Problem was formalized considering properties of the semantic evaluation of `PVS0` programs that were applied in verifying the termination of PVS specifications. The equivalence between predicative and functional evaluation operators was vital to this aim. Furthermore, the compositionality of multiple-function `PVS0` programs, straightforwardly enabled by the possibility of calling different functions, makes it easy to formalize of properties such as Turing Completeness. Therefore, enriching the model was an important design decision to simplify the mechanization of this property and the theorems mentioned above.

Keywords: Turing Completeness, Halting Problem, Rice's Theorem, Fixed-Point Theorem, Post Correspondence Problem, Word Problem, Undecidability

Contents

Chapter 1

Introduction

This work aims to formalize the theory of models of computation developed to assist the static analysis of source code. Such models have been mainly applied to check termination. As it is known, termination is an undecidable problem. Despite this fact, it is possible to design a program, that receives another program as input, and attempts to check if it halts, does not halt, or that answers “do not know”. This possibility has given rise to a variety of efforts to design tools to check and automate the analysis of termination in computer science. Indeed, since 2004, there is a tournament called *Annual International Termination Competition* in which the substantial progress of tools for this goal can be verified.

Rust is an example of a language that contains mechanisms for “termination checking” (Payet et al. [2022]). The Rust language provides several mechanisms for it, which help prevent bugs and ensure code safety. One of these mechanisms is borrow checking, which prevents a variable from being used after it has been moved to another variable. In addition, Rust also has lifetime checking rules that ensure references are not used after the lifetime of the referenced object. Another important mechanism is the use of enums and exhaustive pattern matching, which ensure that all possibilities are considered and handled appropriately. These and other mechanisms make Rust a safe and reliable language for software development.

There are several termination analysis techniques. Blanqui and Koprowski formalized in Coq various terminating techniques used in modern automated provers, generating the CoLoR library and building a program called Rainbow (Blanqui and Koprowski [2011]) a prover program. The prover takes as an input a term rewriting system and outputs a proof tree file in an XML format. The Rainbow program transforms it into a Coq file to certificate it by the CoLoR library. The termination notions used are dependency pairs, dependency graphs, and reduction pairs.

CeTA is a Haskell program extracted from an Isabelle/HOL library called IsaFoR

(Thiemann and Sternagel [2009]). The input of CeTA is an Isabelle/HOL proof tree. In addition to the termination criteria used by Rainbow, CeTA provides a combination of proofs in the dependency pairs framework. Another criterion of termination is the Calling Context Graphs, which were used in implementations of termination verification automation in the ACL2 theorem prover Chamarthi et al. [2011], Manolios and Vroon [2006] as well as in the Prototype Verification System (PVS) Muñoz et al. [2021, 2023].

The proof assistant PVS has been used to formalize the equivalence among different criteria of termination. The considered criteria include the Size-Change termination principle, and the Calling Context Graphs, Matrix Weighted Graphs, and Turing termination approaches Alves Almeida [2021], Alves Almeida and Ayala-Rincón [2020], Avelar [2015], Muñoz et al. [2021, 2023].

The model of computation specified in PVS for this task was designed to have the operational semantics of the PVS specification language. This model is restricted to the first-order fragment of the PVS specification language providing two advantages. First, maintaining the structure of the PVS specification language, it simplifies the analysis of the termination of PVS programs. Second, having simpler grammar simplifies the case analysis required in all formalizations. Indeed, the restriction of the computational model reduces the number of cases to be analyzed in formal proofs.

The model of computation is a recursive first-order language called single-function `PVS0`, or `SF-PVS0`. Using this language, the below criteria of termination were proved equivalent.

- For any input the program computes an output according to the operational semantics of the model - Semantical termination.
- For any input, the program control flow tree, formed by the consecutive recursive calls, is finite - Related to Turing termination.
- For any input, the arguments of any sequence of nested recursive calls can be measured through a well-founded order that decreases after each recursive call - Related to Turing termination.
- Every possible infinite function call sequence (following the program control flow) would cause an infinite descent in some data values - Size-Change Principle

  (Lee et al. [2001]). It can be implemented as:

  - Let G be a digraph such that the vertices represent the *calling contexts*, which are the possible recursive calls together with the conditions to reach each recursive call. In addition, the edges of G represent the possibility to execute the recursive call labeling the head of the edge, consecutively after executing the recursive call labeling the tail of the edge. If there exists a circuit in G that can be executed forever, termination does not hold. Such a circuit corresponds to an infinite function call sequence. To check termination, the *calling context graph* criterion uses a family of measures over the parameters of the calling contexts, and searches for a feasible combination of these measures that strictly decreases over each possible circuit in the graph. - Calling Context Graphs criterion (Manolios and Vroon [2006]).

– Let G be a calling context graph. The family of the measures can be organized in matrices labeling the vertices whose particular operational algebra may indicate decreases over each possible circuit in the graph. - Matrix Weighed Graphs criterionAvelar [2015], Muñoz et al. [2021, 2023].

The above-mentioned formalizations are available in the libraries CCG and PVS0 of the NASA PVS Library and were developed in cooperation between the members of the "Grupo de Teoria da Computação da UnB" and the NASA Langley Formal Methods Team ↗.

In addition to the formalization of the equivalence of termination criteria on `SF-PVS0`, a question to be answered is what kind of properties this computational model holds. The answer is useful to show the adequateness and limits of the `SF-PVS0` computational model. Indeed, PVS allows for the specification of non-necessarily partial recursive functions. For example, it is possible to specify a PVS function that decides the halting problem, but this function is not executable. However, constrained to the `SF-PVS0` model, The undecidability of the halting problem can be proven in PVS (Ferreira Ramos et al. [2018]).

When the model is used to formalize results as Rice's Theorem, the `SF-PVS0` model does not provide an easy mechanism for the composition of functions. However, Rice's Theorem requires it. The grammar of the `SF-PVS0` model contains only one recursive function. Thus, the multiple-function `PVS0`, or `MF-PVS0`, was designed from the `SF-PVS0` to support several recursive functions.

The `MF-PVS0` also needed to be constrained to be equivalent to the class of partial recursive functions. Indeed, `SF-PVS0` and `MF-PVS0` are built over basic operations called built-in operators. If the built-in operators are successor, greater than, bijection from a tuple of naturals into natural, and first and second projections composed to the bijection inverse, is enough to formalize that the `MF-PVS0` is Turing-Complete. The `SF-PVS0` was constrained to be in levels: the built-in operators are arbitrary at level zero but, at level n+1 the built-in operators are the built-in operators from level n together with the terminating `PVS0` functions from level n.

Main contributions

The main contributions of this work are listed below.

- Turing Completeness. The formalization proves that the class of partial recursive `MF-PVS0` programs, built from basic functions and predicates (projections, successor, constants, greater-than), are closed under the operations of composition, minimization, and primitive recursion. It follows the lines of proofs such as the one in (Turing [1937b]) that shows λ-definability of partial recursive functions. For the formalization of this result, some specialized constructions were necessary. For instance, for composition and primitive recursion, since a `MF-PVS0` program receives as argument a natural that represents a tuple of naturals resulting from applications of several `MF-PVS0` programs, it was necessary to construct bijections from tuples of naturals to naturals.

- Rice's Theorem. It was formalized as a corollary of the Recursion Theorem used to build a partial recursive `MF-PVS0` program, which processes its Gödel number. If it is the number of a program that satisfies any semantic property, then the program behaves as if it does not satisfy the property; otherwise, it behaves as if it satisfies it. This formalization follows the classical diagonalization argumentation as done in (Sipser [2012]) for Turing Machines.

- Additional results such as the undecidability of the Halting Problem and the Fixed-Point Theorem were also formalized. The Halting Problem was formalized either for single or multiple-function. There are two versions of the Theorem of the undecidability of the Halting Problem. One says that it is undecidable that a program halts for a specific input (Halting Problem, formalized for `SF-PVS0` and `MF-PVS0`). Another one says that it is undecidable that a program halts for all inputs (Uniform Halting Problem, formalized only for `MF-PVS0`). The latter was proved just as a corollary of Rice's Theorem. The former was proved using diagonalization and arbitrary Gödelizations of partial recursive `MF-PVS0` programs, and a bijection from tuples of naturals to naturals to encode `MF-PVS0` programs and inputs. The formalization follows the proof style in (Sipser [2012]) for Turing Machines. The Fixed-Point Theorem was formalized as a consequence of the fact that it is possible to build the universal `MF-PVS0` program. Besides, it uses a *diagonal* program whose semantics is receiving two arguments: the first one is a program that transforms an input program into another one, and the second one is a value. The *diagonal* program applies the first argument to itself and the second argument. This proof is the only one in the development that uses the bijectivity of the Gödelization of partial recursive `MF-PVS0` programs. The construction follows the proof in (Floyd and Beigel [1994]).

Organization of the document

- Chapter 2 contains a brief introduction to the proof assistant PVS, and the syntax and semantics of the multiple (`MF-PVS0`) and single-function (`SF-PVS0`) `PVS0` computation models.
- Chapter 3 describes the constraints for the `SF-PVS0` and `MF-PVS0` models. In particular, it discusses the specification of partial recursive `PVS0` programs and discusses the formalization of the undecidability of the Halting Problem.
- Chapters 4 and 5 discuss technicalities of the formalization of computational properties of the `MF-PVS0` model. The former chapter discusses the formalization of Turing Completeness of the constrained `MF-PVS0` model. Additionally, it presents the formalization of the Recursion Theorem. The latter chapter discusses the formalization of Rice's Theorem, also presenting a series of corollaries of Rice's Theorem and the Fixed Point Theorem for the multiple-function model.
- Chapter 6 discusses the reduction from the Word Problem for Thue Systems to the Post Correspondence Problem. Furthermore, the discussion highlights the increased difficulty in formalizing the reduction from the Halting Problem for the current model to these problems, compared to a reduction from the Halting Problem for Turing Machines.
- Chapter 7 presents related work.
- Chapter 8 concludes and discusses possible future work.

Chapter 2

Single- and Multiple-Function PVS0 Computational Models

Section 2.1 shortly describes the proof assistant PVS. Section 2.2 discusses the semantics of the single- and multiple-function PVS0 models.

2.1 The proof assistant PVS

The PVS (Prototype Verification System) (Owre et al. [1992]) is a proof assistant based on higher-order logic. It supports the specification of functions and predicates on functions and relations. PVS also supports the definition of recursive functions and inductive predicates. The most careful and detailed description of the semantics of PVS is available in (Owre and Shankar [1999]).

To ensure the correctness of recursive functions specified in PVS, it is necessary to prove that they terminate. A fundamental part of correctness consists in providing termination proofs of such functions. The user should provide a measure over the arguments of the function such that after all recursive calls this measure decreases following a well-founded relation. After supplying the measure, by static analysis, PVS generates proof obligations, presented as lemmas, about the correctness of the types of arguments used in the specification of the operator. These lemmas are called Type Correctness Conditions (TCCs). PVS tries to discharge automatically all TCCs, but if some TCC is not proved, the user should prove it.

For example, the following recursive function multiplies the first `i` members of the input list `l`:

```
mult_l(l : list[nat], i : below[length(l)]) : RECURSIVE posnat =
   IF i = 0 THEN 1
```

```
      ELSE car(l) * mult_l(cdr(l),i-1)
      ENDIF
  MEASURE length(l)
```

In this case, PVS generates a TCC related to termination using the provided measure and the well-founded ordering over naturals:

```
mult_l_TCC3: OBLIGATION
  FORALL (l: list[nat], i: below[length[nat](l)]):
    NOT i = 0 IMPLIES length(cdr(l)) < length(l);
```

TCCs are conditions that must be satisfied to ensure that a function is correctly defined.

One of the key features of PVS is its rich and expressive type system, which allows developers to specify the types of variables, functions, and other constructs in a highly precise and flexible manner. The type system in PVS is based on Church's Simple Type Theory extended with subtypes, dependent types and datatypes. The basic types are, for example, `bool`, `int`, `rational`, and `real`. From the basic types, more complex types can be built. For example, the type of functions from the domain `T1` to the image `T2` is denoted by `[T1 -> T2]`.

Subtyping is the feature of converting predicates into types. Subtyping allows PVS to model complex relationships between types. This enables developers to write more flexible and reusable code and reduces the risk of errors caused by type mismatches. Subtyping works as follows: let `Pred?` be a predicate over the type `T`. To transform it into a type for the variable or constant `x` it is enough to write `x : (Pred?)`. PVS also allows for renaming types, creating synonyms for more complex built-in PVS types. For example, if the PVS user writes `P : TYPE = (Pred?)`, the type `P` is the same as the type `Pred?`. Subtyping predicates in PVS are well described in (Rushby et al. [1998]).

PVS makes use of the datatype mechanism to produce theories that introduce operations for constructing, recognizing, and accessing datatype expressions, define structural recursion schemes over datatype expressions, and assert axioms such as those for extensionality and induction.

For example, the datatype below is about the `MF-PVS0` model.

```
mf_PVS0Expr[T:TYPE+] : DATATYPE
BEGIN
    % constants
   cnst(get_val:T) : cnst? : mf_PVS0Expr
   % variable
```

```
    vr : vr? : mf_PVS0Expr
     % unary operators
    op1(get_op:nat,get_arg:mf_PVS0Expr) : op1? : mf_PVS0Expr
    % binary operators
     op2(get_op:nat,get_arg1,get_arg2:mf_PVS0Expr) : op2? : mf_PVS0Expr
    % recursive call
    rec(get_from_list: nat, get_arg:mf_PVS0Expr) : rec? : mf_PVS0Expr
    % if-then-else
     ite(get_cond,get_if,get_else:mf_PVS0Expr) : ite? : mf_PVS0Expr

END mf_PVS0Expr
```

PVS includes a powerful type checker that automatically verifies that expressions are well-typed, and can also infer sometimes the types of variables and other constructs based on their context. However, type-checking inference in PVS is undecidable. It is undecidable because it is possible to specify the type `P` for a semantic predicate, but deciding a semantic predicate is impossible by Rice's Theorem. Therefore, the user sometimes must supply proof about typing.

The PVS-proof engine works through Gentzen's calculus. Gentzen's calculus is a calculus of sequents, in which logical consequences are represented by sequents, which consist of an antecedent and a consequent, which are lists of formulas. The goal is to prove that the conjunction of the formulas in the antecedent has as consequence the disjunction of formulas in the consequent. The calculus works by applying rules of proof in the sequent, transforming it into a simple equivalent sequent, or splitting it into two or more sequents.

For example, using the command (`case` ϕ) PVS splits the current sequent into two sequents. In one of them, the formula ϕ is into the antecedent and in the other sequent it is added to the consequent. In the example below, $\Gamma \vdash \Delta$ is a sequent, where Γ is the antecedent and Δ is the consequent.

$$\frac{\phi, \Gamma \vdash \Delta \qquad \Gamma \vdash \phi, \Delta}{\Gamma \vdash \Delta} \text{(case } \phi\text{)}$$

Note that in the right branch, the formula ϕ is in the consequent. It occurs because the sequent $\Gamma \vdash \phi, \Delta$ is equivalent to $\neg\phi, \Gamma \vdash \Delta$. This equivalence is called c-equivalence.

Another example of a PVS command is (`inst` fnum "a") for instantiation. It replaces the variable by "a" in a formula numbered as fnum quantified using `FORALL` in the antecedent or by `EXISTS` in the consequent.

$$\frac{\phi[x/a], \Gamma \vdash \Delta}{\forall x\phi, \Gamma \vdash \Delta} \text{(inst -1 "}a\text{")} \qquad\qquad \frac{\Gamma \vdash \phi[x/a], \Delta}{\Gamma \vdash \exists x\phi, \Delta} \text{(inst 1 "}a\text{")}$$

The command (skolem) allows to skolemize universal quantifiers in the consequent and existential quantifiers in the antecedent.

$$\frac{\Gamma \vdash \phi[x/x_1], \Delta}{\Gamma \vdash \forall x \phi, \Delta}\text{(skolem)} \qquad \frac{\phi[x/x_1], \Gamma \vdash \Delta}{\exists x \phi, \Gamma \vdash \Delta}\text{(skolem)}$$

The command (flatten) transforms the formulas of the form $\neg\phi$ or $\psi \wedge \phi$ in the antecedent, and $\psi \vee \phi$, $\psi \rightarrow \phi$ or $\neg\phi$ in the consequent as follows:

- if the formula $\neg\phi$ is in the antecedent (consequent), eliminates it and includes the formula ϕ in the consequent (antecedent);
- if the formula $\psi \wedge \phi$ is in the antecedent, it is eliminated, and the formulas ψ and ϕ included in the antecedent;
- if the formula $\psi \vee \phi$ is in the consequent, it is eliminated, and the formulas ψ and ϕ included in the consequent;
- if the formula $\psi \rightarrow \phi$ is in the consequent, it is eliminated, and the formula ψ is included in the antecedent, and ϕ is included in the consequent.

The command (split) takes the formulas of the form $\psi \vee \phi$ or $\psi \rightarrow \phi$ in the antecedent, and $\psi \wedge \phi$ in the consequent splitting the sequent into simple sequents as follows:

- if the formula $\psi \vee \phi$ is in the antecedent, the sequent is split into two new sequents; one of them includes ψ in the antecedent and the other includes ϕ in the antecedent;
- if the formula $\psi \rightarrow \phi$ is in the antecedent, the sequent is split into two sequents; one includes ψ in the consequent and the other includes ϕ in the antecedent;
- if the formula $\psi \wedge \phi$ is in the consequent, the sequent is split into two sequents; one sequent includes ψ and the other ϕ in the consequent.

Induction is a powerful technique used in mathematical proofs to establish the truth of a statement for all possible cases by proving it for a base case and then for an arbitrary case assuming it is true for some smaller cases. In PVS, induction is a fundamental concept used to prove properties of datatypes, functions, and other mathematical objects. PVS provides a built-in tactic called “induction” that applies structural induction on a specified term or formula. This tactic automatically generates subgoals for the base case and the inductive steps accordingly to the datatype over which induction is applied.

For example, the auxiliary lemma below was formalized by induction on the datatype of programming expressions `ms_PVS0Expr`, a `MF-PVS0` expression structure. The recursive function `offset_rec` adds an offset to the recursive call indexes of `MF-PVS0` expressions.

The lemma states that the composition of the `offset_rec` by m and n is the same that a unique application of this function by $m + n$.

```
offset_composition : LEMMA
FORALL(expr : mf_PVS0Expr, m,n : nat):
(offset_rec(m) o offset_rec(n))(expr) = offset_rec(m+n)(expr)
```

As we will see immediately, `MF-PVS0` expressions may be constants, variables, built-in unary or binary operators, function calls, or branching instructions. PVS generates an inductive schema that considers all cases involved in the datatype. For instance, it generates the case below for branching instruction expressions of the form `ite(expr1, expr2, expr3)`. Such expressions are if-then-else instructions with its guard (`expr1`), then expression (`expr2`) and else expression (`expr3`).

```
FORALL ( expr1: mf_PVS0Expr[Val],  expr2: mf_PVS0Expr[Val],
           expr3: mf_PVS0Expr[Val]):
    ((FORALL (m, n: nat):
        (offset_rec(m) o offset_rec(n))(expr1) =
         offset_rec(m + n)(expr1))
      AND
      (FORALL (m, n: nat):
         (offset_rec(m) o offset_rec(n))(expr2) =
          offset_rec(m + n)(expr2))
       AND
       FORALL (m, n: nat):
         (offset_rec(m) o offset_rec(n))(expr3) =
          offset_rec(m + n)(expr3))
     IMPLIES
     FORALL (m, n: nat):
       (offset_rec(m) o offset_rec(n))
           (ite(expr1 , expr2 , expr3))
        = offset_rec(m + n)(ite(expr1, expr2, expr3))
```

2.2 Specification of the Single- and Multiple-function `PVS0` model

The following grammar describes the first-order functional language called single-function `PVS0` (`SF-PVS0`). This is specified in PVS as the datatype `PVS0Exp`

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

The following grammar describes the first-order functional language called multiple-function PVS0 (MF-PVS0). It is specified in PVS as the datatype mf_PVS0Expr.

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

The interpretation of the expressions given by the above grammars is over an uninterpreted type T. Thus, the evaluation (or interpretation) will consider the inputs and outputs of type T. Constants of the type T are represented as cnst(T), and vr represents a PVS0 variable. More precisely, in an expression under evaluation (*expr*) the symbol vr is used to indicate where the argument of the function represented by the expression is applied. Note that the variable is unique, i.e., the expressions represent functions that contain only one argument. Expressions of the form op1($\mathbb{N}$, *expr*) represent the application of unary operators, chosen from a list of unary operators according to the (natural) index given as the first argument, on the result of the evaluation of the second argument, *expr*. Expressions of the form op2($\mathbb{N}$, *expr*, *expr*) are similar, but for binary operators.

The grammars above differ only in the recursion case. Expressions of the form rec(*expr*) and rec($\mathbb{N}$, *expr*) represent recursive calls. For the case of the evaluation of an expression in the SF-PVS0 model, the evaluation of rec(*expr*) will interpret the result of the evaluation of *expr* and recursively evaluate the main expression. For the case of the MF-PVS0 model, as for unary and binary operators, the index argument in rec($\mathbb{N}$, *expr*) references an expression in a list of expressions that are to be called in the evaluation.

Finally, expressions of the form ite(*expr*, *expr*, *expr*) represent branching *if-then-else* instructions. For the evaluation of branching expressions, an element $\perp$ from T is selected and distinguished to represent *False* in the evaluation of ite guards. If the evaluation of the first expression of ite, that is its guard, is different from $\perp$, it is interpreted as *True*, and the evaluation returns the evaluation of the second argument of the ite expression; otherwise, it returns the evaluation of the third ite expression.

The evaluation of expressions requires a 4-tuple $\langle O_1, O_2, \perp, E_f \rangle$, where O_1 and O_2 are lists of built-in unary and binary operators, $\perp$ is the element of T selected to interpret *False*, and E_f, the kernel, is a single expression for the case of the SF-PVS0 model, and a non-empty list of expressions for the case of the MF-PVS0 model. Such 4-tuples are the *programs* of the SF-PVS0 and MF-PVS0 model. In a MF-PVS0 program, $\langle O_1, O_2, \perp, E_f \rangle$, the *main function* is the first expression in its kernel, E_f. Both `SF-PVS0` and `MF-PVS0` programs are specified in PVS with the same type name, respectively, `PVS0` and `PVS0`. PVS can distinguish them by context. However, in this document, the multiple and single-function `PVS0` programs and expressions are called `MF-PVS0` and `SF-PVS0`, respectively. The only difference between objects of these types appears in the expression part, either a single function or a list of functions, respectively.

For distinguishing `SF-PVS0` programs in this document from `MF-PVS0`, the kernel is written in lower-case letters as $\langle O_1, O_2, \perp, e_f \rangle$, where e_f emphasizes the kernel is a unique recursive function, and the kernel of a `MF-PVS0` program is written as upper-case E_f in $\langle O_1, O_2, \perp, E_f \rangle$ and represents a non-empty list of `MF-PVS0` expression.

PVS provides a suite of structures to specify a `MF-PVS0` program: finite sequences, sets, indexed functions, etc. However, using lists helps to better instance a concrete `PVS0` program. Finite lists, of length $n \in \mathbb{N}$, are represented as: $[a_0, \cdots, a_{n-1}]$, As usual, for a list L, the operators $|L|$, and $L(i)$, respectively, gives the *length* of the list and selects the i^{th} element, for any $i < |L|$. The *tail* of a non-empty list L is denoted by $cdr(L)$, and the concatenation of the lists L_1 and L_2 by $L_1 :: L_2$. Mapping of lists by a function f is denoted as $map(f)(L)$.

To give semantics to the single and `MF-PVS0` expressions, predicates and functions were implemented. The semantic predicate ε was implemented as an inductive predicate. It is specified in a polymorphic way as the predicates `semantic_rel_expr` and `semantic_rel_expr`, for the single-function and multiple-function models, respectively. Using inductive predicates does not require proving their termination. It is a crucial design decision that supports the analysis of non-terminating `PVS0` program specifications, which correspond to (non-terminating) partial recursive functions.

Differently from the inductive predicates, implementing recursive functions in PVS requires that the user provides a measure over the arguments such that each recursive call decreases this measure according to a well-founded relation. Therefore, the semantic evaluation function χ (specified in PVS for the single- and multiple-function model respectively as `eval_expr` and `eval_expr`) has a counter as an argument. Whenever the function χ evaluates a recursive call in an expression, the counter decreases. When the counter reaches zero, the function χ returns a special value $\diamondsuit$, which means that the evaluation of the `PVS0` expression is not possible.

The evaluation predicates are shown in the tables 2.1 and 2.2 and the evaluation functions are shown in table 2.3 and 2.4.

In the predicate ε, the variables v_i and v_o are the input and output of the evaluation of the expression e respectively. In the evaluation, the expression e matches each case of the grammar, executing an action of the interpretation. For example, in case the expression

Table 2.1: SF-PVS0 program evaluation predicate - $pvs_0 = \langle O_1, O_2, \bot, e_f \rangle$ (`semantic_rel_expr` ↗)

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

is a constant symbol (`cnst`(v)), the output must be the interpretation of this symbol. If the expression is either an application of unary, binary function (`op1` or `op2`) or the multiple-function recursive call (`rec`), the interpretation verifies if the sub-expressions have outputs, and in case they do, the interpretation must search using the index j the operator or the expression in the list of operators or of expressions applying it to the outputs.

In case the predicate ε evaluates a single-function recursive call (`rec`), it must consider only e_f as the recursive function.

The function χ (table 2.3) works similarly to the predicate ε. The output type of the function χ is $T \cup \{\Diamond\}$. In PVS, it is implemented using the functor `Maybe`(T) where an element from this type may be either `none` (represented by the $\Diamond$) or `Some`(t) (where $t \in T$). Having two manners of semantic evaluation gives more flexibility in the formalization.

The evaluation predicate and function are equivalent, i.e., whenever the evaluated expression produces an output for a specific input, the results must be the same. It is expressed by the following lemma.

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

Table 2.2: MF-PVS0 program evaluation predicate - $pvs_0 = \langle O_1, O_2, \bot, E_f \rangle$ (semantic_rel_expr ↗)

```
$\varepsilon(pvs_0)(e, v_i, v_o)$  :=  CASES $e$ OF
          cnst($v$) :  $v_o = v$;
                 vr :  $v_o = v_i$;
      op1($j, e_1$) :  $\exists\,(v' : T)$ :
                       $\varepsilon(pvs_0)(e_1, v_i, v') \wedge$
                       IF $j < |O_1|$ THEN $v_o = O_1(j)(v')$
                       ELSE $v_o = \bot$;
 op2($j, e_1, e_2$) :  $\exists\,(v', v'' : T)$ :
                       $\varepsilon(pvs_0)(e_1, v_i, v') \wedge$
                       $\varepsilon(pvs_0)(e_2, v_i, v'') \wedge$
                       IF $j < |O_2|$ THEN $v_o = O_2(j)(v', v'')$
                       ELSE $v_o = \bot$;
      rec($j, e_1$) :  $\exists\,(v' : T) : \varepsilon(pvs_0)(e_1, v_i, v') \wedge$
                       IF $j < |E_f|$ THEN
                          $\varepsilon(pvs_0)(E_f(j), v', v_o)$
                       ELSE $v_o = \bot$;
ite($e_1, e_2, e_3$) :  $\exists\,(v' : T) : \varepsilon(pvs_0)(e_1, v_i, v') \wedge$
                       IF $v' \neq \bot$ THEN $\varepsilon(pvs_0)(e_2, v_i, v_o)$
                       ELSE $\varepsilon(pvs_0)(e_3, v_i, v_o)$.
```

The necessity and sufficiency of the above property are formalized, both for the SF-PVS0 and the MF-PVS0 model, respectively, as lemmas semantic_rel_eval_expr ↗ (SF-PVS0 model) and the semantic_rel_eval_expr ↗(MF-PVS0 model), and the lemmas eval_expr_semantic_rel ↗ (SF-PVS0 model) and eval_expr_semantic_rel ↗(MF-PVS0 model).

In an evaluation of a MF-PVS0 program, the first expression to be evaluated is the head of the kernel, which is expressed by the predicate γ. Let $\langle O_1, O_2, \bot, E_f \rangle$ be a PVS0 program which is abbreviated by $\langle E_f \rangle$. The predicate γ is defined as:

$$\gamma\langle E_f \rangle(v_i, v_o) := \varepsilon\langle E_f \rangle(E_f(0), v_i, v_o) \tag{2.1}$$

The projection of n-tuple is given by:

$$\langle A_1, \cdots, A_i, \cdots, A_n \rangle' i := A_i \tag{2.2}$$

An important property to be analyzed is termination. Termination of programs may have two versions: the program outputs an answer for a specific input and the program always outputs an answer for all inputs. Termination is expressed by the polymorphic predicates T_ε and T_χ.

Table 2.3: SF-PVS0 program evaluation function - $pvs_0 = \langle O_1, O_2, \bot, e_f \rangle$ (eval_expr ↗)

```
$\chi(pvs_0)(n, e, v_i)$  :=
  IF $n = 0$ THEN $\diamondsuit$ ELSE  CASES $e$ OF
     cnst($v$) :  $v$;
           vr  :  $v_i$;
    op1($j, e_1$) :  IF $j < |O_1|$ THEN
                       LET $v' = \chi(pvs_0)(n, e_1, v_i)$ IN
                       IF $v' = \diamondsuit$ THEN $\diamondsuit$ ELSE $O_1(j)(v')$
                     ELSE $\bot$;
 op2($j, e_1, e_2$) :  IF $j < |O_2|$THEN
                       LET $v' = \chi(pvs_0)(n, e_1, v_i)$,
                           $v'' = \chi(pvs_0)(n, e_2, v_i)$ IN
                       IF $v' = \diamondsuit \lor v'' = \diamondsuit$ THEN $\diamondsuit$
                       ELSE $O_2(j)(v', v'')$
                     ELSE $\bot$;
       rec($e_1$) :  LET $v' = \chi(pvs_0)(n, e_1, v_i)$ IN
                     IF $v' = \diamondsuit$ THEN $\diamondsuit$
                     ELSE $\chi(pvs_0)(n - 1, e_f, v')$
 ite($e_1, e_2, e_3$) :  LET $v' = \chi(pvs_0)(n, e_1, v_i)$ IN
                     IF $v' = \diamondsuit$ THEN $\diamondsuit$
                     ELSE IF $v' \neq \bot$ THEN
                                        $\chi(pvs_0)(n, e_2, v_i)$
                     ELSE $\chi(pvs_0)(n, e_3, v_i)$.
```

$$T_\varepsilon(\langle e_f \rangle, v_i) := \exists\,(v_o : T)\;:\;\gamma\langle e_f \rangle(v_i, v_o) \tag{2.3}$$

$$T_\varepsilon\langle e_f \rangle := \forall\,(v : T)\;:\;T_\varepsilon(\langle e_f \rangle, v) \tag{2.4}$$

$$T_\chi(\langle e_f \rangle, v_i) := \exists\,(n : \mathbb{N})\;:\;\chi\langle e_f \rangle(n, e_f, v_i) \neq \diamondsuit \tag{2.5}$$

$$T_\chi\langle e_f \rangle := \forall\,(v : T)\;:\;\chi\langle e_f \rangle(n, e_f, v_i) \neq \diamondsuit \tag{2.6}$$

$$T_\varepsilon(\langle E_f \rangle, v_i) := \exists\,(v_o : T)\;:\;\gamma\langle E_f \rangle(v_i, v_o) \tag{2.7}$$

$$T_\varepsilon\langle E_f \rangle := \forall\,(v : T)\;:\;T_\varepsilon(\langle E_f \rangle, v) \tag{2.8}$$

$$T_\chi(\langle E_f \rangle, v_i) := \exists\,(n : \mathbb{N})\;:\;\chi\langle E_f \rangle(n, E_f(0), v_i) \neq \diamondsuit \tag{2.9}$$

$$T_\chi\langle E_f \rangle := \forall\,(v : T)\;:\;\chi\langle E_f \rangle(n, E_f(0), v_i) \neq \diamondsuit \tag{2.10}$$

The definitions for SF-PVS0 program termination 2.3, 2.4, 2.5 and 2.6 are specified as determined? ↗, terminating? ↗, eval_expr_n? ↗, and eval_expr_termination

Table 2.4: MF-PVS0 program evaluation function - $pvs_0 = \langle O_1, O_2, \bot, E_f \rangle$ (`eval_expr` ↗)

```
$\chi(pvs_0)(n, e, v_i)$  :=
  IF $n = 0$ THEN $\diamondsuit$ ELSE CASES $e$ OF
     cnst($v$) : $v$;
            vr : $v_i$;
     op1($j, e_1$) : IF $j < |O_1|$ THEN
                       LET $v' = \chi(pvs_0)(n, e_1, v_i)$ IN
                       IF $v' = \diamondsuit$ THEN $\diamondsuit$ ELSE $O_1(j)(v')$
                     ELSE $\bot$;
     op2($j, e_1, e_2$) : IF $j < |O_2|$THEN
                       LET $v' = \chi(pvs_0)(n, e_1, v_i)$,
                           $v'' = \chi(pvs_0)(n, e_2, v_i)$ IN
                       IF $v' = \diamondsuit \vee v'' = \diamondsuit$ THEN $\diamondsuit$
                       ELSE $O_2(j)(v', v'')$
                     ELSE $\bot$;
     rec($j, e_1$) : LET $v' = \chi(pvs_0)(n, e_1, v_i)$ IN
                     IF $v' = \diamondsuit$ THEN $\diamondsuit$
                     ELSE IF $j < |E_f|$ THEN
                            $\chi(pvs_0)(n - 1, E_f(j), v')$
                     ELSE $\bot$;
     ite($e_1, e_2, e_3$) : LET $v' = \chi(pvs_0)(n, e_1, v_i)$ IN
                     IF $v' = \diamondsuit$ THEN $\diamondsuit$
                     ELSE IF $v' \neq \bot$ THEN
                                 $\chi(pvs_0)(n, e_2, v_i)$
                     ELSE $\chi(pvs_0)(n, e_3, v_i)$.
```

↗, respectively.

The definitions for MF-PVS0 program termination 2.7, 2.8, 2.9 and 2.10 are specified respectively as `determined?` ↗ `determined?`, `terminating?` ↗, `eval_expr_n?` ↗, and `eval_expr_termination` ↗.

The need for the composition of the PVS0 programs was required to prove Rice's Theorem, but it was not necessary for formalizing the Halting Problem.

The Halting Problem was formalized for the SF-PVS0 programs. In such a computational model, program composition is not straightforward. In the formalization of this theorem for the single-function model, only terminating programs were composed (cf. Ferreira Ramos et al. [2018]). Thus, upgrading the model to the multiple-function version simplified the formalizations of Rice's Theorem, the recursion theorem, and the fixed-point theorem, among other properties.

Composing MF-PVS0 programs requires only that they share the same built-in functions and the same interpretation of false. In addition, an adjustment of the indices of the recursive calls is also necessary. This adjustment is a simplification of the offset operation used by assembly languages, where calls for pieces of code shift them and add an offset to the labels of these pieces. This offset addition in `MF-PVS0` programs is expressed by the function $\_^{+}\_$ (`offset_rec(_)(_)` ↗):

$$\begin{array}{rcl} e^{+n} & := & \texttt{CASES}\ e\ \texttt{OF} \\ \texttt{cnst}(v) & : & \texttt{cnst}(v); \\ \texttt{vr} & : & \texttt{vr}; \\ \texttt{op1}(j, e_1) & : & \texttt{op1}(j, e_1^{+n}); \\ \texttt{op2}(j, e_1, e_2) & : & \texttt{op2}(j, e_1^{+n}, e_2^{+n}); \\ \texttt{rec}(j, e_1) & : & \texttt{rec}(j+n, e_1^{+n}); \\ \texttt{ite}(e_1, e_2, e_3) & : & \texttt{ite}(e_1^{+n}, e_2^{+n}, e_3^{+n}) \end{array} \tag{2.11}$$

The function $\_^{+}\_$ is also used in a polymorphic way to sum an offset to a list of `MF-PVS0` expressions, as it as below:

$$L^{+n} := map(\_^{+n})(L) \tag{2.12}$$

Using the offset operator, the correctness of the composition of two `MF-PVS0` programs $\langle O_1, O_2, \bot, A\rangle$ and $\langle O_1, O_2, \bot, B\rangle$, in short, written as $\langle A\rangle$ and $\langle B\rangle$, respectively, is expressed by the property:

$$\begin{array}{l} \forall(v_i, v_o) : \exists(v) : \gamma\langle B\rangle(v_i, v) \wedge \gamma\langle A\rangle(v, v_o) \Leftrightarrow \\ \gamma\langle[\texttt{rec}(1, \texttt{rec}(1+|A|, \texttt{vr}))] :: A^{+1} :: B^{+(1+|A|)}\rangle(v_i, v_o) \end{array}$$

The code above applies B to the variable, and after that, applies A, resulting in a composition.

An example of functions that can be composed in `MF-PVS0` model but in `SF-PVS0` is not straightforward is the functions f and g.

$$succ(n) := n+1 \qquad greater(m, n) := \texttt{IF}\ m > n\ \texttt{THEN}\ 1\ \texttt{ELSE}\ 0 \tag{2.13}$$

$$O_1 := [succ] \qquad O_2 := [greater] \tag{2.14}$$

$$F := [\texttt{ite}(\texttt{op2}(0, \texttt{vr}, \texttt{cnst}(0)), \texttt{vr}, \texttt{rec}(0, \texttt{vr}))] \qquad f := \langle O_1, O_2, 0, F\rangle \tag{2.15}$$

$$G := [\texttt{ite}(\texttt{op2}(0, \texttt{vr}, \texttt{cnst}(0)), \texttt{op1}(0, \texttt{vr}), \texttt{rec}(0, \texttt{vr}))] \qquad g := \langle O_1, O_2, 0, G\rangle \tag{2.16}$$

The composition of the program f and g is given by:

$$\langle [\texttt{rec}(1, \texttt{rec}(1 + |A|, \texttt{vr}))] :: F^{+1} :: G^{+(1+|F|)} \rangle \tag{2.17}$$

To prove Turing completeness and Rice's Theorem, it is necessary to formalize some lemmas about shift code. As previously, consider `MF-PVS0` programs $\langle O_1, O_2, \perp, A\rangle$ and $\langle O_1, O_2, \perp, B\rangle$. The first lemma is:

Lemma 1 (Shift code - `add_rec_list_aux`).

$$\forall(\langle A\rangle, \langle B\rangle, e, v_i, n) \ : \ \chi\langle B\rangle(n, e, v_i) = \chi\langle A :: B^{+|A|}\rangle(n, e^{+|A|}, v_i)$$

This lemma means that in an evaluation of the expression e considering the `MF-PVS0` program $\langle B\rangle$, it is possible to concatenate a list A in front of B without changing the evaluation semantics, adjusting the indices accordingly in `rec` expressions contained by e and B.

The definitions are about valid indices:

$$\begin{aligned} &\mathit{valid\_index\_rec}(e, n) := \\ &\forall(i, e_1) : \\ &\quad \mathit{subterm}(\texttt{rec}(i, e_1), e) \Rightarrow i < n \end{aligned} \tag{2.18}$$

$$\begin{aligned} &\mathit{valid\_index}(E_f) := \\ &\forall(i < |E_f|) : \\ &\quad \mathit{valid\_index\_rec}(E_f(i), |E_f|) \end{aligned} \tag{2.19}$$

The second lemma is:

Lemma 2 (Shift code - `add_rec_list_aux2`).

$$\begin{aligned} &\forall(\langle B\rangle, v_i, n) : \\ &\forall(\langle A\rangle \mid \mathit{valid\_index}(A)) : \forall(e \mid \mathit{valid\_index\_rec}(e, |A|)) : \\ &\qquad\qquad\qquad \chi\langle A\rangle(n, e, v_i) = \chi\langle A :: B\rangle(n, e, v_i) \end{aligned}$$

This lemma is similar to Lemma 1. Still, the indices of the `rec` expressions in the evaluated expression e and the list A of the `MF-PVS0` program $\langle A\rangle$ must be valid references to a `MF-PVS0` expression in A. The list B of the `MF-PVS0` program $\langle B\rangle$ 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 natural and (sub)expressions. The type of pair (n, e) is $\mathbb{N} \times$ `PVS0Expr`, where `PVS0Expr` is the type of `PVS0` expressions.

The following results are formalized over the `MF-PVS0` programs model: Undecidability of the Halting Problem, Fixed-Point Theorem, Turing Completeness, Recursion Theorem, and Rice‘s Theorem.

Chapter 3

Undecidability of the Halting Problem for Single- and Multiple-Function PVS0 Programs

The undecidability of the Halting Problem was initially mechanized for the SF-PVS0 model (Ferreira Ramos et al. [2018]). Then, it was formalized for the MF-PVS0 computational model specified in mf_pvs0_halting_problem_undecidability, as reported in (Ramos et al. [2022]).

3.1 Computable and Recursive PVS0 Programs

The primary goal of the library PVS0 was the formalization of equivalence among termination criteria for the SF-PVS0 model. This work aims to prove that such a class of programs is a reasonable computational model. For this, initially, we proved that the SF-PVS0 model satisfies classical properties as the undecidability of the Halting Problem. Moreover, proving fundamental properties as Turing completeness of SF-PVS0 programs presented difficulties since this model does not naturally support the composition of programs, a fundamental property of recursive functions. The undecidability of the Halting Problem was formalized for the SF-PVS0 model using a construction of the composition possible exclusively for terminating programs. Nevertheless, results such as Turing completeness, recursion, fixed-point, and Rice's theorems require the composition of non-terminating programs. In contrast to single-function programs, multiple-function programs allow for a natural construction of the composition of two programs, which resumes concatenating their lists of function expressions and adequately updating the indices used in the recursive calls.

There are two significant instances of the Halting Problem. One of them states that it is impossible to program a halting verifier for any program and one input. The other states that it is impossible to program such a verifier for any program with any input. The former problem is known as the Halting problem, and the latter problem is known as the Uniform Halting Problem. This chapter discusses the formalization of the Halting Problem. The Uniform Halting Problem was mechanized as a corollary 1 of Rice's Theorem and will be discussed in section 5.1. Initially, we will provide the necessary preliminary definitions and constraints on the types of the `MF-PVS0` programs. Indeed, note that in the 4-tuple that defines `MF-PVS0` programs, the lists of unary and binary operators can contain any operator. In particular, since PVS allows for the definition of non-computable functions, this implies that building programs that decide termination, or another type of Oracle, is possible. Because of this, `MF-PVS0` programs must be constrained to behave as partial recursive functions.

The input/output type is the natural numbers. Natural numbers are used to encode `SF-PVS0` and `MF-PVS0` programs and their inputs through a Gödelization function.

The classes of partial recursive `SF-PVS0` and `MF-PVS0` programs were specified in the theories `pvs0_computable` and `mf_pvs0_computable`. In both theories, the list of unary and binary operators, O_1 and O_2, and the element to interpret as false, $\perp$, are parameters of the theories. Let $\boldsymbol{O_1}$ and $\boldsymbol{O_2}$ be arbitrary built-in operators that represent the parameters of the theories.

Let the bijection from a tuple of naturals to naturals be given by κ_2 (`tuple2nat`):

$$\kappa_2(m,n) := \frac{(m+n+1)(m+n)}{2} + n \tag{3.1}$$

The inverse function κ_2^{-1} is implemented recursively:

$$\begin{aligned} \kappa_2^{-1}(i) \;\; := \;\; & \texttt{IF } \; i = 0 \texttt{ THEN } (0,0) \\ & \texttt{ELSE IF } \; \kappa_2^{-1}(i-1)'1 = 0 \texttt{ THEN } (\kappa_2^{-1}(i-1)'2+1, 0); \\ & \texttt{ELSE } \;\; (\kappa_2^{-1}(i-1)'1 - 1, \kappa_2^{-1}(i-1)'2+1) \end{aligned} \tag{3.2}$$

The partial recursive single-function programs are constrained in levels. The level zero contains single-function programs with basic built-in operators, and higher levels contain single-function programs that use built-in operators built from a terminating program from the previous level. The reason for this restriction, is that if unrestricted built-in operator were allowed, it would be possible to specify a non-computable function that decides the halting problem. The levels are specified as:

$$\begin{array}{l}
\mathit{pvs0\_level}(n)\langle O_1, O_2, \bot, e_f\rangle := \\
\quad \texttt{IF } n = 0 \texttt{ THEN } O_1 = \boldsymbol{O_1} \ \wedge O_2 = \boldsymbol{O_2} \\
\quad \texttt{ELSE} \ (\ \exists p' : \ \mathit{pvs0\_level}(n-1)(p') \wedge \\
\qquad\qquad \texttt{LET } \langle O_1', O_2', \bot, e_f'\rangle = p', \ l_1' = |O_1{}'| \texttt{ IN} \\
\qquad\qquad |O_1| = l_1' + 1 \wedge \\
\qquad\qquad (\ \forall i \in \mathbb{N} : i < l_1' \Rightarrow O_1(i) = O_1{}'(i) \ ) \wedge \\
\qquad\qquad (\ \forall v \in \mathbb{N} : \varepsilon(p')(e_f{}', v, O_1(l_1')(v)) \ ) \ ) \wedge \\
\qquad (\ \exists p' : \ \mathit{pvs0\_level}(n-1)(p') \wedge \\
\qquad\quad \texttt{LET } \langle O_1', O_2', \bot', e_f'\rangle = p', l_2' = |O_2{}'| \texttt{ IN} \\
\qquad\quad |O_2| = l_2' + 1 \wedge \\
\qquad\quad (\ \forall i \in \mathbb{N} : i < l_2' \Rightarrow O_2(i) = O_2{}'(i) \ ) \wedge \\
\qquad\quad (\ \forall v_1, v_2 \in \mathbb{N} : \varepsilon(p')(e_f{}', \kappa_2(v_1, v_2), O_2(l_2')(v_1, v_2)) \ ) \ ),
\end{array} \tag{3.3}$$

The class of partial recursive SF-PVS0 programs is given by:

$$\texttt{partial\_recursive} := \{\mathit{pvs_0} \mid \exists n : \mathit{pvs0\_level}(n)(\mathit{pvs_0})\} \tag{3.4}$$

Note that a `partial_recursive` SF-PVS0 program in level n becomes an operator in level $n+1$ if it is terminating; in this manner, the composition of terminating programs is made by getting operators from the previous level.

The necessary Gödelization for the formalization of the halting problem is expressed by the following lemma and theorem:

Lemma 3. For all level n, there exists a PVS function κ_T from $\mathbb{N}$ to SF-PVS0 programs of level n that is surjective.

The formalization is done by induction in the level of the programs. The induction basis, for level zero programs, uses a bijective function to give this enumeration of expressions:

$$\begin{array}{rcl}
\kappa_{e0}(e) & := & \texttt{CASES } \ e \texttt{ OF} \\
\texttt{vr} & & : 0; \\
\texttt{cnst}(n) & & : n \times 5 + 1; \\
\texttt{rec}(e_1) & & : \kappa_{e0}(e_1) \times 5 + 2; \\
\texttt{op1}(i, e_1) & & : \kappa_2(i, \kappa_{e0}(e_1)) \times 5 + 3; \\
\texttt{op2}(i, e_1, e_2) & & : \kappa_2(i, \kappa_2(\kappa_{e0}(e_1), \kappa_{e0}(e2))) \times 5 + 4; \\
\texttt{ite}(e_1, e_2, e_3) & & : \kappa_2(\kappa_{e0}(e_1), \kappa_2(\kappa_{e0}(e_2), \kappa_{e0}(e_3))) \times 5 + 5
\end{array} \tag{3.5}$$

The inverse function κ_{e0}^{-1} is defined as:

$$
\begin{array}{lll}
\kappa_{e0}^{-1}(n) & := & \\
\texttt{IF} & n = 0 & \texttt{THEN vr} \\
\texttt{ELSE IF} & 5|(n-1) & \texttt{THEN cnst}(\frac{n-1}{5}) \\
\texttt{ELSE IF} & 5|(n-2) & \texttt{THEN rec}(\kappa_{e0}^{-1}(\frac{n-2}{5})) \\
\texttt{ELSE IF} & 5|(n-3) & \texttt{THEN op1}(\kappa_2^{-1}(\frac{n-3}{5})'1, \kappa_{e0}^{-1}(\kappa_2^{-1}(\frac{n-3}{5})'2)) \\
\texttt{ELSE IF} & 5|(n-4) & \texttt{THEN op2}(\kappa_2^{-1}(\frac{n-4}{5})'1, \\
& & \quad \kappa_{e0}^{-1}(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n-4}{5})'2))'1, \\
& & \quad \kappa_{e0}^{-1}(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n-4}{5})'2)'2)) \\
\texttt{ELSE} & 5|(n-5) & \texttt{ite}(\kappa_{e0}^{-1}(\kappa_2^{-1}(\frac{n-5}{5})'1), \\
& & \kappa_{e0}^{-1}(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n-5}{5})'2))'1, \\
& & \kappa_{e0}^{-1}(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n-5}{5})'2)'2))
\end{array}
\tag{3.6}
$$

Then, the SF-PVS0 programs from level zero are enumerated by:

$$\kappa_{P_0}(m) := \langle \boldsymbol{O_1}, \boldsymbol{O_2}, \perp, \kappa_{e0}^{-1}(m) \rangle$$

For the induction step, for programs whose level, n, is greater than zero, assume the function $\kappa_{P_{n-1}}$ exists; thus, the function $\kappa_{T_{n-1}}$ exists such that is a surjective function from naturals to terminating programs of the level $n-1$. To obtain the function, an oracle function is used because the class of the teminating functions is not enumerable. Then, the following function is also surjective, where the function *choose* chooses an element from a non-empty set:

$$
\begin{array}{lll}
\kappa_{P_n}(m) & := \texttt{LET} & \\
& p_1 & = \kappa_2^{-1}(\kappa_2^{-1}(m)'1)'1, \\
& p_2 & = \kappa_2^{-1}(\kappa_2^{-1}(m)'1)'2, \\
& p_3 & = \kappa_2^{-1}(m)'2, \\
& f(x) & = \mathit{choose}(\{r \mid \gamma(\kappa_{T_{n-1}}(p_1))(x, r)\}), \\
& g(x, y) & = \mathit{choose}(\{r \mid \gamma(\kappa_{T_{n-1}}(p_2))(\kappa_2(x, y), r)\}) \ \texttt{IN} \\
& \multicolumn{2}{l}{\langle \kappa_{T_{n-1}}(p_1)'1 :: [f], \kappa_{T_{n-1}}(p_2)'2 :: [g], \perp, \kappa_{e0}^{-1}(p_3) \rangle}
\end{array}
$$

Theorem 1. There exists a PVS function κ_P from $\mathbb{N}$ to `partial_recursive` that is surjective.

The surjective function is:

$$\kappa_P(n) := \mathit{choose}(\{f : \mathbb{N} \to (\mathit{pvs0\_level}(\kappa_2^{-1}(n)'1)) \mid \text{surjective?}(f)\})(\kappa_2^{-1}(n)'2)$$

Above a surjective function is chosen from a set of surjective functions from naturals to SF-PVS0 programs of the level $\kappa_2^{-1}(n)'1$ and it is applied to $\kappa_2^{-1}(n)'2$. The lemma 3 proves that the set of surjective functions is not empty.

The terminating `partial_recursive` `SF-PVS0` programs give rise to the subtype of `computable` `PVS0` programs.

$$\texttt{computable} := \{pvs_0 : \texttt{partial\_recursive} \mid T_\varepsilon(pvs_0)\} \tag{3.7}$$

One important question is about how to build the universal `SF-PVS0` program. building them requires the following functions and list of operators:

$$\begin{array}{lcl}
succ(n) & := & n+1 \\
greater(m,n) & := & \texttt{IF}\ \ m > n\ \texttt{THEN}\ \ 1\ \texttt{ELSE}\ \ 0 \\
\pi_1(n) & := & ((\lambda(m,n:\mathbb{N}):m)\circ\kappa_2^{-1})(n) \\
\pi_2(n) & := & ((\lambda(m,n:\mathbb{N}):n)\circ\kappa_2^{-1})(n) \\
g_1(n) & := & \kappa_{P_{\pi_1(n)}}(\pi_1(\pi_2(n)))'1(\pi_2(\pi_2(n)) \\
g_2(n) & := & \kappa_{P_{\pi_1(n)}}(\pi_1(\pi_2(n)))'2(\pi_2(\pi_2(n))) \\
\boldsymbol{O_1} & := & [succ, \pi_1, \pi_2, g_1, g_2] \\
\boldsymbol{O_2} & := & [greater, \kappa_2] \\
\bot & := & 0
\end{array} \tag{3.8}$$

The following abbreviations are used:

$$\begin{array}{l}
succ^S(e) := \texttt{op1}(0,e);\ \pi_1^S(e) := \texttt{op1}(1,e);\ \pi_2^S(e) := \texttt{op1}(2,e); \\
greater^S(e_1,e_2) := \texttt{op2}(0,e_1,e_2);\ \kappa_2^S(e_1,e_2) := \texttt{op2}(1,e_1,e_2) \\
g_1^S(e) := \texttt{op1}(3,e);\ g_2^S(e) := \texttt{op1}(4,e)
\end{array}$$

To implement a universal program, is necessary to implement a cut-off subtraction, the remainder from, and integer division by five. The cut-off subtraction of a pair of naturals encoded as a unique natural can be specified as *sub*:

```
$sub'4 :=$
        LET $i = \pi_1^S($vr$),\ j = \pi_2^S($vr$)$ IN
         [ite($greater^S(i,j)$,
          $succ^S($rec$(\kappa_2^S(i, succ^S(j))))$),
          cnst(0))]
```

The program *sub* is terminating, thus, programs from level 1 can call it a built-in operator.

Let *sub* be the correspondent function. The lists of built-in operators from level 1 are:

$$\begin{array}{lcl} sub(m,n) & := & \texttt{IF}\ \ m > n\ \texttt{THEN}\ m - n\ \texttt{ELSE}\ \ 0 \\ sub_{\pi}(m) & := & sub(\pi_1(m), \pi_2(m)) \\ \boldsymbol{O_1} & := & [succ, \pi_1, \pi_2, g_1, g_2, sub_{\pi}] \\ \boldsymbol{O_2} & := & [greater, \kappa_2, sub] \end{array} \tag{3.9}$$

Abbreviating $sub^S(e_1, e_2) := \texttt{op2}(2, e_1, e_2)$, the remainder by five is implemented as:

$$\begin{array}{l} rem'_54 := \\ \quad [\texttt{ite}(greater^S(\texttt{vr}, \texttt{cnst}(5)), \\ \quad \texttt{rec}(sub^S(\texttt{vr}, \texttt{cnst}(5))), \\ \quad \texttt{vr})] \end{array}$$

The correspondent function rem_5 can be called by a program of level 2. The built-in operators are:

$$\begin{array}{lcl} rem_5(n) & := & n\%5 \\ rem_{5\kappa_2}(m,n) & := & \kappa_2(m,n)\%5 \\ \boldsymbol{O_1} & := & [succ, \pi_1, \pi_2, g_1, g_2, sub_{\pi}, rem_5] \\ \boldsymbol{O_2} & := & [greater, \kappa_2, sub, rem_{5\kappa_2}] \end{array} \tag{3.10}$$

The integer division by five is given by the following program:

$$\begin{array}{l} div'_54 := \\ \quad [\texttt{ite}(greater^S(\texttt{vr}, \texttt{cnst}(5)), \\ \quad succ^S(\texttt{rec}(sub^S(\texttt{vr}, \texttt{cnst}(5)))), \\ \quad \texttt{cnst}(0))] \end{array}$$

The universal SF-PVS0 program is from level 3 and it has the built-in operators:

$$\begin{array}{lcl} div_5(n) & := & \lfloor \frac{n}{5} \rfloor \\ div_{5\kappa_2}(m,n) & := & \lfloor \frac{\kappa_2(m,n)}{5} \rfloor \\ \boldsymbol{O_1} & := & [succ, \pi_1, \pi_2, g_1, g_2, sub_{\pi}, rem_5, div_5] \\ \boldsymbol{O_2} & := & [greater, \kappa_2, sub, rem_{5\kappa_2}, div_{5\kappa_2}] \end{array} \tag{3.11}$$

The kernel of the universal program is:

```
$U'4 :=$
        $\texttt{LET}\ n = \pi_1^S(\texttt{vr}),\ e_f = \pi_1^S(\pi_2^S(\texttt{vr})),\ e = \pi_1^S(\pi_2^S(\pi_2^S(\texttt{vr}))),\ rem_5^S(o) = \texttt{op1}(6, o)$
            $sub^S(o_1, o_2) = \texttt{op2}(2, o_1, o_2),\ div_5^S(o) = \texttt{op1}(7, o)$
            $v_i = \pi_2^S(\pi_2^S(\pi_2^S(\texttt{vr}))), k_4(x, y, z, w) = \kappa_2^S(x, \kappa_2^S(y, \kappa_2^S(z, w)))$
            $k_{p1}(m, i, o) = \texttt{op1}(3, \kappa_2^S(nm, \kappa_2^S(i, o))).\ k_{p2}(m, i, o) = \texttt{op1}(4, \kappa_2^S(m, \kappa_2^S(i, o)))\ \texttt{IN}$
        $[\texttt{ite}(e,$
            $\texttt{ite}(rem_5^S(sub^S(e, \texttt{cnst}(1))),$
                $\texttt{ite}(rem_5^S(sub^S(e, \texttt{cnst}(2))),$
                    $\texttt{ite}(rem_5^S(sub^S(e, \texttt{cnst}(3))),$
                        $\texttt{ite}(rem_5^S(sub^S(e, \texttt{cnst}(4))),$
                            $, k_{p1}(n, \pi_1^S(div_5^S(sub_S(e, \texttt{cnst}(4)))),$
                                $\texttt{rec}(k_4(n, e_f, \pi_2(div_5^S(sub^S(e, \texttt{cnst}(4)))), v_i))),$
                            $\texttt{ite}(\texttt{rec}(k_4(n, e_f, \pi_1(div_5^S(sub^S(n, \texttt{cnst}(5))), v_i))),$
                                $\texttt{rec}(k_4(n, e_f, \pi_2(\pi_1(div_5^S(sub^S(n, \texttt{cnst}(5)))), v_i))),$
                                $\texttt{rec}(k_4(n, e_f, \pi_2(\pi_2(div_5^S(sub^S(n, \texttt{cnst}(5)))), v_i))))),$
                        $\texttt{ite}(\texttt{rec}(k_4(n, e_f, \pi_2(div_5^S(sub^S(e, \texttt{cnst}(3)))), v_i)),$
                            $, k_{p1}(n, \pi_1^S(div_5^S(sub_S(e, \texttt{cnst}(3)))),$
                                $\texttt{rec}(k_4(n, e_f, \pi_2(div_5^S(sub^S(e, \texttt{cnst}(3)))), v_i)))$
                            $k_{p1}(n, \pi_1^S(div_5^S(sub_S(e, \texttt{cnst}(3)))),$
                                $\texttt{rec}(k_4(n, e_f, \pi_2(div_5^S(sub^S(e, \texttt{cnst}(3)))), v_i))))),$
                    $\texttt{rec}(k_4(n, e_f, e_f, \texttt{rec}(k_4(n, e_f, div_5^S(sub^S(e, \texttt{cnst}(2))), v_i))))),$
                $div_5^S(sub^5(n, \texttt{cnst}(1)))),$
            $v_i)]$
```

For the universal program, the following property holds:

Lemma 4. $\gamma(U)(\kappa_2(n, \kappa_2(\kappa_p(n)'3, \kappa_2(\kappa_p(n)'3, v_i))), v_o) \iff \gamma(\kappa_p(n))(v_i, v_o)$

3.2 Undecidability of the Halting Problem

Using the type `computable`, the undecidability of the halting problem is stated in the theorem below.

Theorem 2 (Undecidability of the Halting Problem for `SF-PVS0` ↗)**.** There is no program *oracle* $= \langle O_1, O_2, \perp, e_o\rangle$ of type `computable` such that for all $pvs_0 = \langle O_1', O_2', \perp, e_f\rangle$

of type `partial_recursive` and for all $n \in \mathbb{N}$,

$$T_\varepsilon(\mathit{pvs}_0, n) \text{ if and only if } \neg\varepsilon(\mathit{oracle})(e_o, \kappa_2(\kappa_P(\mathit{pvs}_0), n), \bot).$$

The proof proceeds by assuming the existence of an oracle to derive a contradiction. Suppose there exists a program $\mathit{oracle} = \langle O_1, O_2, \bot, e_o\rangle$ of type `computable` such as the one presented in the statement of the theorem. Then, a program $\mathit{pvs}_0 = \langle O_1', O_2', \bot, e_f\rangle$ can be defined, where $O_1'(k) = O_1(k)$, for $k < |O_1|$, $O_2'(k) = O_2(k)$, for $k < |O_2|$, and

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

The PVS function *choose* returns an arbitrary element from a non-empty set. The sets used in the definitions of O_1' and O_2' are non-empty since *oracle* is `computable` and, therefore, terminating. The program pvs_0 is built in such a way that it belongs to the next level from the level of *oracle.*
Let n be the natural number $\kappa_P(\mathit{pvs}_0)$. The rest of the proof proceeds by case analysis.

- **Case1** : $\varepsilon(\mathit{oracle})(e_o, \kappa_2(n, n), \bot)$. This case holds if and only if $\neg T_\varepsilon(\mathit{pvs}_0, n)$.

  Expanding T_ε one obtains

$$\begin{aligned} \neg\exists(v : \mathbb{N}) : \exists(v_o : \mathbb{N}) :\ & \varepsilon(\mathit{pvs}_0)(\mathtt{op2}(|O_2|, \mathtt{vr}, \mathtt{vr}), n, v_o) \wedge \\ & \mathtt{IF}\ v_o \neq \bot \\ & \mathtt{THEN}\ \varepsilon(\mathit{pvs}_0)(\mathit{rec}(\mathtt{vr}), n, v) \\ & \mathtt{ELSE}\ \varepsilon(\mathit{pvs}_0)(\mathtt{vr}, n, v). \end{aligned} \tag{3.12}$$

  Expanding ε in $\varepsilon(\mathit{pvs}_0)(\mathtt{op2}(|O_2|, \mathtt{vr}, \mathtt{vr}), n, v_o)$ yields

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

  Since $\varepsilon(\mathit{oracle})(e_o, \kappa_2(n, n), \bot)$ holds, $\bot = v_o$. Therefore, Formula (3.12) is equivalent to

$$\neg\exists(v : \mathbb{N}) : \varepsilon(\mathit{pvs}_0)(\mathtt{vr}, n, v). \tag{3.13}$$

  The predicate $\varepsilon(\mathit{pvs}_0)(\mathtt{vr}, n, v)$ holds if and only if $n = v$. Hence, Formula (3.13) states that $\neg\exists(v : \mathbb{N}) : n = v$, where n is a natural number. This is a contradiction.

- **Case2** : $\neg\varepsilon(oracle)(e_o, \kappa_2(n,n), \bot)$**.** This case holds if and only if $T_\varepsilon(pvs_0, n)$. From the equivalence between T_χ and T_ε, $T_\chi(pvs_0, n)$ holds. If the proof starts directly from $T_\varepsilon(pvs_0, n)$, after expanding and simplifying it, $T_\varepsilon(pvs_0, n)$ is obtained once again, which implies that there is not such an n, giving a contradiction. However, since PVS does not accept the definition of a function that enters into such an infinite loop, the solution is to apply the equivalence between T_χ and T_ε. Expanding the definition of T_χ yields

  $$\exists\, m \in \mathbb{N} \,:\, \chi(pvs_0)(m, \texttt{ite}(\texttt{op2}(|O_2|, \texttt{vr}, \texttt{vr}), \texttt{rec}(\texttt{vr}), \texttt{vr}), n) \neq \diamondsuit.$$

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

  $$\left(\begin{array}{l} \texttt{IF } \chi(pvs_0)(m, \texttt{op2}(|O_2|, \texttt{vr}, \texttt{vr}), n) \neq \diamondsuit \texttt{ THEN} \\ \quad \texttt{IF } \chi(pvs_0)(m, \texttt{op2}(|O_2|, \texttt{vr}, \texttt{vr}), n) \neq \bot \texttt{ THEN} \\ \quad\quad \chi(pvs_0)(m, rec(\texttt{vr}), n) \\ \quad\quad \texttt{ELSE } \chi(pvs_0)(m, \texttt{vr}, n) \\ \texttt{ELSE } \diamondsuit \end{array}\right) \neq \diamondsuit. \tag{3.14}$$

  If the condition of the first if-then-else were false, then Formula (3.14) reduces to $\diamondsuit \neq \diamondsuit$, which is a contradiction. Therefore, this condition must be true. After expanding and simplifying χ, $\chi(pvs_0)(m, \texttt{op2}(|O_2|, \texttt{vr}, \texttt{vr}), n)$ reduces to

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

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

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

  This is a contradiction since $n = \kappa_P(pvs_0)$.

  Thus, $\chi(pvs_0)(m, \texttt{op2}(|O_2|, \texttt{vr}, \texttt{vr}), n) \neq \bot$. Then, Formula (3.14) can be simplified to

  $$\chi(pvs_0)(m, rec(\texttt{vr}), n) \neq \diamondsuit.$$

  Finally, expanding χ results in $\chi(pvs_0)(m-1, e_f, n) \neq \diamondsuit$. This contradicts the minimality of m, completing the proof.

□

As in the former theorem, the formalization of the halting problem for partial recursive `MF-PVS0` programs uses Cantor's diagonalization. The class is restricted to `MF-PVS0` programs in which all indices of the expressions in their kernels, E_f, are valid; i.e., the indices are smaller than the length of the kernel. Let *subterm* be sub-expression relation. Valid indices are specified as below.

$$\begin{array}{l} \mathit{valid\_index\_rec}(e,n) := \\ \forall (i, e_1): \\ \quad \mathit{subterm}(\mathtt{rec}(i,e_1), e) \Rightarrow i < n \end{array} \tag{3.15}$$

$$\begin{array}{l} \mathit{valid\_index}(E_f) := \\ \forall (i < |E_f|): \\ \quad \mathit{valid\_index\_rec}(E_f(i), |E_f|) \end{array} \tag{3.16}$$

The predicate below defines a partial recursive MF-PVS0 program *pvs₀*.

$$\begin{aligned} \mathit{partial\_recursive?}(\mathit{pvs_0}) \;:=\; & \mathit{pvs_0}'1 = O_1 \;\wedge\; \mathit{pvs_0}'2 = O_2 \wedge \\ & \mathit{pvs_0}'3 = \bot \;\wedge \mathit{valid\_index}(\mathit{pvs_0}'4) \end{aligned} \tag{3.17}$$

An important feature of PVS is to use predicates as sub-types. Thus, the predicate *partial_recursive?* is converted to a sub-type of the MF-PVS0 programs type. This sub-type is called `partial_recursive`. From the predicate *partial_recursive?*, the predicate *computable?* expresses the classes of partial recursive MF-PVS0 programs but that is terminating.

$$\mathit{computable?}(\mathit{pvs_0}) := \mathit{partial\_recursive?}(\mathit{pvs_0}) \;\wedge T_\varepsilon(\mathit{pvs_0}) \tag{3.18}$$

This predicate above is converted into a type `computable`. `partial_recursive` and `computable` are polymorphic types, i.e., their definitions depend if they are about SF-PVS0 or MF-PVS0 programs.

From a `partial_recursive` SF-PVS0 program, there exists a `partial_recursive` MF-PVS0 program that behaves the same. Such a partial recursive program is built embedding the operators accumulated in the SF-PVS0 levels, transforming them into an expression in the kernel of a MF-PVS0 program. Thus, the MF-PVS0 model simulates the SF-PVS0 model. It is not formalized, but it is interesting for future work. The converse simulation, i.e., simulating MF-PVS0 programs using SF-PVS0, requires that the SF-PVS0 program receives a natural number that represents a pair with the Gödel number of the MF-PVS0 program and its input. Then, the SF-PVS0 program interprets the Gödel number as a program.

Another option is to show that it is possible to define (the Godelization of) a trace for the computation of a partial recursive function in SF-PVS0 model and then show that a partial recursive function can be computed by searching for a valid trace of the computation of the function for the given input.

The Gödelization of encoding of MF-PVS0 programs together to its input uses the bijection from a tuple of naturals to natural, κ_2. The Gödelization of MF-PVS0 programs is given by the injective function κ_p. This function (`p_recursive2nat`) is a parameter of the theory `mf_pvs0_halting` ↗. The oracle program that decides termination is non-computable is defined using the function κ_2^{-1}:

$$\begin{aligned} O_1 &:= [\lambda(i)(\texttt{LET }(p,n) = \kappa_2^{-1}(i)\texttt{IF }T_\varepsilon(\kappa_p(p),n)\texttt{THEN }\top\texttt{ELSE }\bot] \\ O_2 &:= [] \\ E_f &:= [\texttt{op1}(0,\texttt{vr})] \end{aligned} \tag{3.19}$$

$$\mathit{oracle} := \langle O_1, O_2, \bot, E_f\rangle \tag{3.20}$$

Using this `MF-PVS0` program the following theorem is formalized ↗:

$$\begin{gathered} \forall(\mathit{pvs_0}, n) : \\ (\neg\gamma\langle E_f\rangle(\kappa_2(\kappa_p(\mathit{pvs_0}),n),\bot)) \text{ if and only if } T_\varepsilon(\mathit{pvs_0}, n) \end{gathered} \tag{3.21}$$

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

Theorem 3 (Undecidability of the Halting Problem for `MF-PVS0` ↗)**.** For all O_1, O_2, $\bot$, and κ_p, there is no program *oracle* of type `computable` such that for all $\mathit{pvs_0} = \langle O_1, O_2, \bot, E_f\rangle$ of type `partial_recursive` and for all $n \in \mathbb{N}$,

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

The classical formalization of the undecidability of the halting problem starts by assuming the existence of an oracle capable of deciding whether a program halts for an input. A Gödelization function transforms the tuple of the program and input into a single input to the oracle. After that, using the oracle, another program is created such that if the encoded program halts it enters into an infinite loop. Otherwise, it produces an answer and halts. Passing this program as an input to itself results in the expected contradiction. In the proof of undecidability, any representation as a natural number of a specific `MF-PVS0` program is built using the supposed oracle. This is the reason for using a general Gödelization function.
The formalization needs the assumption that the function κ_2 belongs to the list of binary operators:

$$\exists(k < |O_2|) : O_2(k) = \kappa_2$$

The main difference between theorems 2 and 3 is that in the latter, the lists of unary and binary operators and the false element are fixed, and the PVS0 program $\mathit{pvs_0}$ must be the program below. This will give rise to a contradiction.

$$\langle \boldsymbol{O_1}, \boldsymbol{O_2}, \bot, [\mathtt{ite}(\mathtt{rec}(1, \mathtt{op2}(k, \mathtt{vr}, \mathtt{vr})), \mathtt{rec}(0, \mathtt{vr}), \mathtt{cnst}(\top))] :: \mathit{map}(\beta(1))(\mathit{oracle}'4)\rangle$$

Where i is the index in the list of binary operators to the codification of a tuple of naturals to naturals.
Considering the case $\gamma(\mathit{oracle})(\kappa_2(\kappa_L(\mathit{pvs_0}), n), \bot)$, $\neg T_\varepsilon(\mathit{pvs_0}, n)$ is simplified to $\neg\exists(v : \mathbb{N}) : \varepsilon(\mathit{pvs_0})(\mathtt{cnst}(\top), n, v)$. which reduces to as a contradiction.
Considering the case $\neg\gamma(\mathit{oracle})(\kappa_2(\kappa_L(\mathit{pvs_0}), n), \bot)$, let $n = \kappa_L(\mathit{pvs_0})$. The supposition $\neg\gamma(\mathit{oracle})(\kappa_2(n, n), \bot)$ is equivalent to $T_\varepsilon(\mathit{pvs_0}, n)$. Expanding it and applying the equivalence with T_χ:
$\exists\, m \in \mathbb{N} \,:\, \chi(\mathit{pvs_0})(m, \mathtt{ite}(\mathtt{rec}(1, \mathtt{op2}(k, \mathtt{vr}, \mathtt{vr})), \mathtt{rec}(0, \mathtt{vr}), \mathtt{cnst}(\top)), n) \neq \diamondsuit$.
If exists a natural m that makes the previous formula hold, suppose that is the minimal.
Expanding χ, yields:

$$\left(\begin{array}{l}\mathtt{IF}\ \chi(\mathit{pvs_0})(m, \mathtt{rec}(1, \mathtt{op2}(k, \mathtt{vr}, \mathtt{vr})), n) \neq \diamondsuit\ \mathtt{THEN}\\ \quad \mathtt{IF}\ \chi(\mathit{pvs_0})(m, \mathtt{rec}(1, \mathtt{op2}(k, \mathtt{vr}, \mathtt{vr})), n) \neq \bot\ \mathtt{THEN}\\ \quad\quad \chi(\mathit{pvs_0})(m, \mathit{rec}(0, \mathtt{vr}), n)\\ \quad\quad \mathtt{ELSE}\ \chi(\mathit{pvs_0})(m, \mathtt{vr}, n)\\ \mathtt{ELSE}\ \diamondsuit\end{array}\right) \neq \diamondsuit. \tag{3.22}$$

If $\chi(\mathit{pvs_0})(m, \mathtt{rec}(1, \mathtt{op2}(k, \mathtt{vr}, \mathtt{vr})), n) \neq \diamondsuit$ does not hold, the previous if-then-else statement reduce to $\diamondsuit$, that is a contradiction.
Expanding $\chi(\mathit{pvs_0})(m, \mathtt{rec}(1, \mathtt{op2}(k, \mathtt{vr}, \mathtt{vr})), n) \neq \bot$, it reduces to:

$$\chi(\mathit{pvs_0})(m-1, \mathit{oracle}(0), \kappa_2(n, n)) \neq \bot \tag{3.23}$$

That is equivalent to $\neg\gamma(\mathit{oracle})(\kappa_2(\kappa_L(\mathit{pvs_0}), n), \bot)$.
Thus, the formula 3.22 is simplified to $\chi(\mathit{pvs_0})(m, \mathit{rec}(0, \mathtt{vr}), n) \neq \diamondsuit$. Expanding it:

$$\chi(\mathit{pvs_0})(m-1, \mathtt{ite}(\mathtt{rec}(1, \mathtt{op2}(k, \mathtt{vr}, \mathtt{vr})), \mathtt{rec}(0, \mathtt{vr}), \mathtt{cnst}(\top)), n) \neq \diamondsuit.$$

That contradicts the minimality of m.

□

Note that both formalizations require the application of the equivalence between the termination predicates T_ε and T_χ when the case $\neg\gamma(\mathit{oracle})(\kappa_2(\kappa_L(\mathit{pvs_0}), n), \bot)$ holds is analyzed. This equivalence is necessary because if only T_ε is applied, it will generate an infinite expansion of the predicate trying to find witnesses of the existential quantification used in the specification of T_ε.

Chapter 4

Formalization of the computational properties of the `PVSO` Model - Turing Completeness, and Recursion Theorem

This and the next chapters present the technicalities of the formalization in PVS of computational properties of the `MF-PVSO` model in detail. Section 4.1 describes the constraints necessary to formalize that the `MF-PVSO` model is closed for composition, minimization, and primitive recurrence. Additionally, the constant, successor, and projection can be implemented using the model, concluding that it is Turing Complete. In the second part of the chapter, Section 4.2 discusses the formalization of the Recursion Theorem.

4.1 Turing Completeness of `MF-PVSO` Model

Turing Completeness of the `MF-PVSO` model depends on the built-in operators used. For example, if the lists of the built-in operators are empty, and the domain and range are natural numbers, the model is not Turing Complete. One criterion to be Turing Complete is to simulate the Partial Recursive Function working: the constant, successor, and projection must be implemented, and the model must be closed under primitive recurrence, minimization, and composition.

First of all, the built-in operators, the lists of unary and binary operators, and the element ($\perp$) that plays the role of "false" are given in Equation 4.1

$$\begin{array}{lcl} \mathit{succ}(n) & := & n+1 \\ \mathit{greater}(m,n) & := & \texttt{IF}\ \ m>n\ \texttt{THEN}\ \ 1\ \texttt{ELSE}\ \ 0 \\ \pi_1(n) & := & ((\lambda(m,n:\mathbb{N}):m)\circ\kappa_2^{-1})(n) \\ \pi_2(n) & := & ((\lambda(m,n:\mathbb{N}):n)\circ\kappa_2^{-1})(n) \\ \boldsymbol{O_1} & := & [\mathit{succ},\pi_1,\pi_2] \\ \boldsymbol{O_2} & := & [\mathit{greater},\kappa_2] \\ \bot & := & 0 \end{array} \tag{4.1}$$

In short, `MF-PVS0` programs of the form $\langle \boldsymbol{O_1}, \boldsymbol{O_2}, \bot, E_f\rangle$ will be written simply as $\langle E_f\rangle$. These fixed built-in operators are the same used in the formalization of the Recursion Theorem (see Section 4.2). The main idea in the formalization of Turing Completeness is to build `MF-PVS0` programs that process their Gödel Number. Furthermore, using these built-in operators the Turing Completeness of the model is guaranteed. We will use the predicate below (that does not belong to the specification) for the class of `MF-PVS0` programs having $\boldsymbol{O_1}, \boldsymbol{O_2}$ and 0 as parameters.

$$\begin{array}{lllll} \mathit{partial\_recursive?}(\mathit{pvs_0}) := & \mathit{pvs_0}'1 = \boldsymbol{O_1} & \wedge & \mathit{pvs_0}'2 = \boldsymbol{O_2} & \wedge \\ & \mathit{pvs_0}'3 = \bot & \wedge & \mathit{valid\_index}(\mathit{pvs_0}'4) & \end{array}$$

Any `MF-PVS0` program $\mathit{pvs_0}$ that belongs to the above predicate is said to be of type `partial_recursive` (PVS polymorphism enables using the same name as for the `SF-PVS0` model). This type is obtained as an instantiation of the parameters of the type `partial_recursive`. In the specification, to define the type `partial_recursive` it is enough to pass the above parameters to the theory `mf_pvs0_computable`.

In order to show Turing completeness of the class of `MF-PVS0` programs of such type, it is only necessary to prove that there are implementations of the constant, successor, and projection functions and that the class is closed under composition, minimization, and primitive recurrence. The interesting cases in this formalization are those related to the projection implementation and the proofs of closure under composition, minimization, and primitive recurrence.

The n-tuples in PVS are specified as lists of naturals but encoded in the formalization as a unique natural. The function *nat2list* transforms uniquely a natural m into a list of naturals of length n. See below.

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

The following abbreviations are used:

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

For a MF-PVS0 program $\mathit{pvs_0}$ of type `partial_recursive`, the focus would be on the list of expressions, i.e., on $\mathit{pvs_0}'4$. The MF-PVS0 `partial_recursive` program *equal*, specified below, verifies if the pair of naturals encoded as a unique natural are equal.

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

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

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

$$\begin{array}{l} \mathit{proj\_aux}'4 := \\ \qquad \texttt{LET}\ i = \pi_1^S(\texttt{vr}),\ j = \pi_1^S(\pi_2^S(\texttt{vr})),\ k = \pi_1^S(\pi_2^S(\pi_2^S(\texttt{vr}))), \\ \qquad\quad l = \pi_2^S(\pi_2^S(\pi_2^S(\texttt{vr}))), k_4(x,y,z,w) = \kappa_2^S(x, \kappa_2^S(y, \kappa_2^S(z,w)))\ \texttt{IN} \\ \qquad\ [\texttt{ite}(\mathit{greater}^S(j,i), \\ \qquad\qquad \texttt{rec}(0, k_4(\mathit{succ}^S(i), j, k, \pi_2^S(l))), \\ \qquad\qquad \texttt{ite}(\texttt{rec}(1, \kappa_2^S(j,k)), l, \pi_1^S(l)))] :: \mathit{equal}'4^{+1} \end{array}$$

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

$$\begin{array}{l} \mathit{proj}'4 := \\ \qquad \texttt{LET}\ i = \pi_1^S(\texttt{vr}),\ j = \pi_1^S(\pi_2^S(\texttt{vr})),\ k = \pi_2^S(\pi_2^S(\texttt{vr})), \\ \qquad\quad k_4(x,y,z,w) = \kappa_2^S(x, \kappa_2^S(y, \kappa_2^S(z,w)))\ \texttt{IN} \\ \qquad\ [\texttt{rec}(1, k_4(\texttt{cnst}(0), i, j, k))] :: \mathit{proj\_aux}'4^{+1} \end{array}$$

The correctness of the `MF-PVS0` `partial_recursive` program for projection, *proj*, is formalized as Lemma 5 using γ as given in the formula (2.1). The lemma shows that *proj* projects correctly the i-th element of any tuple encoded as the natural m (seen as an $n+1$ tuple).

Lemma 5 (Correctness of Projection - `proj_correctness` ↗)**.**

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

The analysis of composition requires the functions *exprComp* and *chainOffset* below. In these functions, l is a non-empty list of say m list of expressions that are the kernel of `MF-PVS0` programs. The idea is to simulate the composition of an m-ary function with m functions. As can be observed in Chapter 2.2, the composition of two `MF-PVS0` programs of the same class of partial recursive functions is straightforward. Nevertheless, to show Turing completeness, the composition must be specified between a `MF-PVS0` program and an m-tuple of `MF-PVS0` programs. To specify an n-tuple of an arbitrary length, non-empty lists are used.

```
exprComp(n, l)  :=
        IF  $|l| = 1$ THEN rec($n$, vr);
        ELSE  $\kappa_2^S$(rec($n$, vr), $\mathit{exprComp}(n + |l(0)|, \mathit{cdr}(l))$)
```

```
chainOffset(n, l)  :=
        IF  $|l| = 1$ THEN $l(0)^{+n}$;
        ELSE  $l(0)^{+n} :: \mathit{chainOffset}(n + |l(0)|, \mathit{cdr}(l))$;
```

Let F be a `MF-PVS0` program, L a non-empty list of `MF-PVS0` programs, and $l := \mathit{map}(\lambda(x,y,z,w) : w)(L)$. To specify composition, a new list of `MF-PVS0` expressions is created, where the head of this new list is a recursive expression that calls the expression $F'4$, and the expressions in the tail are given by l. In addition, the function *chainOffset* (`chain_offset` ↗) adjusts the indices of the `MF-PVS0` expressions of F and L in the composition. When evaluating this new list of expressions, i.e., the new `MF-PVS0` program, the function *exprComp* (`expr_comp` ↗) generates a `MF-PVS0` expression whose evaluation codifies a list of naturals (which are the results of the application of the `MF-PVS0` programs in L to the input) into a natural. This natural is then passed as an input parameter to evaluate F (*comp* ↗).

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

Finally, the composition lemma also requires a way to represent n-tuples of naturals (formalized as non-empty list of naturals) into naturals (*list2nat*):

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

Now, it is possible to establish the correctness of the composition lemma for a `MF-PVS0` kernel F' and a list of kernels l as follows.

Lemma 6 (Correctness of Composition - `comp_is_composition`)**.**

$$\begin{aligned} &\forall (F, l \mid |l| > 0) : \forall (v_i, v_o) : \\ &\ \gamma(\mathit{comp}(F'4, l))(v_i, v_o) \Leftrightarrow \\ &\ \exists (l_n \mid |l_n| = |l|) : \\ &\quad \forall (i \mid i < |l_n|) : \gamma\langle l(i)\rangle(v_i, l_n(i)) \wedge \\ &\quad \gamma(F)(\kappa_2(|l_n|, \mathit{list2nat}(l_n)), v_o) \end{aligned}$$

The formalization of the correctness of composition (Lemma 6) is by induction on the length of l. The proof requires some technical lemmas, such as showing that the indices of function calls used by `rec`, generated by the functions *exprComp* and *chainOffset*, are valid. This guarantees that *comp* generates a `MF-PVS0` `partial_recursive` program.

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

$$\begin{aligned} &\mathit{min\_aux}(F'4)'4 := \\ &[\texttt{ite}(\texttt{rec}(1, \texttt{vr}), \\ &\qquad \texttt{rec}(0, \kappa_2^S(\mathit{succ}^S(\pi_1^S(\texttt{vr})), \pi_2^S(\texttt{vr}))), \\ &\qquad \pi_1^S(\texttt{vr}))] :: F'4^{+1} \end{aligned}$$

Using *min_aux*, the (list of expressions of the) minimization of F is specified below (*min*).

$$\mathit{min}(F'4)'4 := \ [\texttt{rec}(1, \kappa_2^S(\texttt{cnst}(0), \texttt{vr}))] :: \mathit{min\_aux}(F'4)'4^{+1}$$

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

Lemma 7 (Correctness of Minimization - `min_correctness`).

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

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

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

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

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

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

$$
\begin{aligned}
&prim\_recur(recur'4, final'4)'4 := \\
&\qquad \texttt{LET}\ \ i = \pi_1^S(\texttt{vr}), \\
&\qquad\qquad j = \pi_2^S(\texttt{vr}), \\
&\qquad\qquad less_1(x) = \texttt{rec}(1 + |recur'4| + |final'4|, x), \\
&\qquad\qquad recur\_fun(x, y, z) = \texttt{rec}(1, \kappa_2^S(x, \kappa_2^S(y, z))), \\
&\qquad\qquad final\_fun(x) = \texttt{rec}(1 + |recur'4|, x), \\
&\qquad\qquad recur\_call(x, y) = \texttt{rec}(0, \kappa_2^S(x, y)) \\
&\qquad \texttt{IN} \\
&\qquad\quad [\texttt{ite}(i, \\
&\qquad\qquad recur\_fun(recur\_call(less_1(i), j), less_1(i), j), \\
&\qquad\qquad final\_fun(j))] :: \\
&\qquad\quad recur'4^{+1} :: final^{+1+|recur'4|} :: \\
&\qquad\quad sub1'4^{1+|recur'4|+|final'4|}
\end{aligned}
$$

The function *prim_recur* receives two kernels of `partial_recursive` programs, *recur*'4 and *final*'4, and returns another `partial_recursive` program that implements primitive recurrence for the respective associated functions r and f as below, where ρ is primitive recurrence operator.

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

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

Lemma 8 (Primitive Recurrence Correctness - `prim_recur_correctness`).

$$\begin{array}{l}\forall(\mathit{recur},\mathit{final}):\forall(i,j,v_o):\\ \gamma(\mathit{prim\_recur}(\mathit{recur}'4,\mathit{final}'4))(\kappa_2(i,j),v_o)\ \Leftrightarrow\\ \exists(l_n\mid i+1=|l_n|):v_o=l_n(|l_n|-1)\wedge\\ \ \gamma(\mathit{final})(j,l_n(0))\wedge\\ \ \forall(k\mid k<|l_n|-1):\\ \ \ \gamma(\mathit{recur})(\kappa_2(l_n(k),\kappa_2(k,j)),l_n(k+1))\end{array}$$

As with the correctness of composition (Lemma 6), the formalization of the correctness of minimization and primitive recursion (Lemmas 7 and 8) requires additional technical elements such as the inductive predicates below that avoid expansions of the predicate γ resulting in expansions of the evaluation predicate ε.

$$\begin{array}{l}\mathit{min\_relation}(i,j,F,v_o)\ \ :=\\ \qquad\qquad \texttt{IF}\ \ \gamma(F)(\kappa_2(i,j),0)\ \texttt{THEN}\ v_o=i\\ \qquad\qquad \texttt{ELSE IF}\ \ \exists(k):\gamma(F)(\kappa_2(i,j),k)\\ \qquad\qquad\quad \texttt{THEN}\ \ \mathit{min\_relation}(i+1,j,F,v_o)\\ \qquad\qquad \texttt{ELSE}\ \ False.\end{array}$$

$$\begin{array}{l}\mathit{prim\_recur\_relation}(\mathit{recur},\mathit{final})(i,j)(v_o)\ \ :=\\ \qquad\qquad \texttt{IF}\ \ i\neq 0\ \texttt{THEN}\ \ \exists(z):\\ \qquad\qquad\ \ \gamma(\mathit{recur})(\kappa_2(z,\kappa_2(i-1,j)),v_o)\wedge\\ \qquad\qquad\ \ \mathit{prim\_recur\_relation}(\mathit{recur},\mathit{final})(i-1,j)(z)\\ \qquad\qquad \texttt{ELSE}\ \ \gamma(\mathit{final})(j,v_o).\end{array}$$

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

The sufficiency of the correctness of minimization is formalized using the inductive schema given by the predicate *min_relation*. The necessity is formalized using the equivalence between the evaluation function χ and the predicate ε. As discussed in Chapter 2.2, the function χ provides the measure to be applied in inductive proofs like the one performed for formalizing the necessity. Similarly, the sufficiency of the correctness of the primitive recurrence is formalized using the predicate *prim_recur_relation*, while a straightforward induction on i proves necessity.

4.2 Recursion Theorem

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

The formalization uses the technique of building a virus program as explained in (Sipser [2012]). Instead of replicating itself, the `MF-PVS0` program processes its Gödel number. The formalization uses the same basic operators for the successor, projection, greater-than, and the bijection κ_2 operators applied to formalize Turing Completeness for the `MF-PVS0` model. We dispose of constructions obtained in the proof of Turing Completeness such as composition, minimization, and primitive recurrence. However, in the formalization, we opt for building the required constructions implementing `MF-PVS0` programs directly. These programs are designed using simultaneously several `MF-PVS0` programs simplifying in this manner the constructions. The result is specified as theorem 4.

The Gödel number is calculated according to the following function.

$$\begin{array}{rcl}\kappa_e(len)(e) & := & \texttt{CASES } e \texttt{ OF} \\ \texttt{vr} & : & 0; \\ \texttt{cnst}(v) & : & v \times 5 + 1; \\ \texttt{rec}(j, e_1) & : & (j + \kappa_e(len)(e_1) \times (len + 1)) \times 5 + 2; \\ \texttt{op1}(j, e_1) & : & \kappa_2(j, \kappa_e(len)(e_1)) \times 5 + 3; \\ \texttt{op2}(j, e_1, e_2) & : & \kappa_2(j, \kappa_2(\kappa_e(len)(e_1), \kappa_e(len)(e_2))) \times 5 + 4; \\ \texttt{ite}(e_1, e_2, e_3) & : & \kappa_2(\kappa_e(len)(e_1), \kappa_2(\kappa_e(len)(e_2), \kappa_e(len)(e_3))) \times 5 + 5 \end{array} \tag{4.2}$$

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

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

$$\begin{array}{rcl} rdc(l) & := & reverse(cdr(reverse(l))) \\ \alpha_{aux}(l) & := & \texttt{IF } |l| = 1 \texttt{ THEN } l(0) \\ & & \texttt{ELSE } \kappa_2(\alpha_{aux}(rdc(l)), l(|l| - 1)) \\ \alpha(l) & := & \texttt{IF } |l| = 0 \texttt{ THEN } 0 \\ & & \texttt{ELSE } \kappa_2(|l| - 1, \alpha_{aux}(l)) + 1 \end{array} \tag{4.3}$$

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

The Gödelization from the class of partial recursive functions to naturals, that uses α, is given as below.

$$\kappa_p(pvs_0) := \alpha(map(\kappa_e(|pvs_0{}'4| - 1))(pvs_0{}'4)) - 1 \tag{4.4}$$

The function κ_p (`p_recursive2nat`) Gödelizes the `partial_recursive` `MF-PVS0` programs.

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

The function κ_e^{-1} is defined as below.

$$\begin{array}{lll}
\kappa_e^{-1}(lim)(n) \;:= & & \\
\quad\texttt{IF} & n=0 & \texttt{THEN vr} \\
\quad\texttt{ELSE IF} & 5|(n-1) & \texttt{THEN cnst}(\frac{n-1}{5}) \\
\quad\texttt{ELSE IF} & 5|(n-2) & \texttt{THEN rec}(\frac{n-2}{5}\%(lim+1), \kappa_e^{-1}(lim)(\lfloor\frac{n-2}{5\times(lim+1)}\rfloor)) \\
\quad\texttt{ELSE IF} & 5|(n-3) & \texttt{THEN op1}(\kappa_2^{-1}(\frac{n-3}{5})'1, \kappa_e^{-1}(lim)(\kappa_2^{-1}(\frac{n-3}{5})'2)) \\
\quad\texttt{ELSE IF} & 5|(n-4) & \texttt{THEN op2}(\kappa_2^{-1}(\frac{n-4}{5})'1, \\
 & & \qquad \kappa_e^{-1}(lim)(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n-4}{5})'2))'1, \\
 & & \qquad \kappa_e^{-1}(lim)(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n-4}{5})'2)'1)) \\
\quad\texttt{ELSE} & 5|(n-5) & \texttt{ite}(\kappa_e^{-1}(lim)(\kappa_2^{-1}(\frac{n-4}{5})'1), \\
 & & \kappa_e^{-1}(lim)(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n-4}{5})'2))'1, \\
 & & \kappa_e^{-1}(lim)(\kappa_2^{-1}(\kappa_2^{-1}(\frac{n-4}{5})'2)'1))
\end{array} \tag{4.5}$$

Above, $a|b$ means a divides b, $a\%b$ is the remainder of the division of a by b and $\lfloor\frac{a}{b}\rfloor$ is the floor of the division of a and b.

Below, the inverse functions α_{aux}^{-1} and α^{-1} are implemented.

$$\begin{array}{rcl}
\alpha_{aux}^{-1}(len, n) & := & \texttt{IF } len = 0 \texttt{ THEN } [n]; \\
 & & \texttt{ELSE } \alpha_{aux}^{-1}(len-1, \kappa_2^{-1}(n)'1) :: [\kappa_2^{-1}(n)'2] \\
\alpha^{-1}(n) & := & \texttt{IF } n = 0 \texttt{ THEN } []; \\
 & & \texttt{ELSE } \alpha_{aux}^{-1}(\kappa_2^{-1}(n-1)'1, \kappa_2^{-1}(n-1)'2)
\end{array} \tag{4.6}$$

The specification of two inverses require two arguments: $\alpha_{aux}^{-1}(len, n)$ and $\kappa_e^{-1}(len)(n)$. In the function α_{aux}^{-1}, len is a natural that defines the length $(len-1)$ of the list of naturals encoded by n. In the function κ_e^{-1}, the argument len is the length of the indices of the `rec` expressions, and n encodes a `MF-PVS0` expression.

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

The inverse of κ_p is specified below.

$$\kappa_p^{-1}(n) := \langle map(\kappa_e^{-1}(|\alpha^{-1}(n+1)|-1))(\alpha^{-1}(n+1))\rangle$$

Lemma 9 (Invertibility of the functions κ_e and α_{aux} - `PVS02nat_nat2PVS0_limit`, `nat2PVS0_PVS02nat_limit`, `nat2listnat_aux_cons2nat` and conversion lemma `cons2nat_nat2listnat_aux`)**.** Left and right invertibility of the operators κ_e and α_{aux} is formalized as:

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

The Gödelization function allows specifying the Recursion Theorem (Kleene's second Recursion Theorem).

Theorem 4. [Recursion Theorem - `Recursion_Theorem`]

$$\begin{array}{l} \forall(E_f) : \exists(print : \texttt{partial\_recursive}) : \\ \quad \texttt{LET}\quad self = \langle E_f :: print'4^{+|E_f|}\rangle \ \ \texttt{IN} \\ \qquad\qquad partial\_recursive?(self)\ \wedge \\ \qquad \forall(i)\ : \varepsilon(self)(print'4(0)^{+|E_f|}, i, \kappa_p(self)) \end{array}$$

To build *print*, the same idea of programming computing viruses is followed. A list of expressions to calculate the Gödel number of *self* is added to E_f. In this manner, one guarantees the desired behavior of *self* that is to be able to calculate its own Gödel number, as a quine does, but also to process it accordingly to the programmers desire. Thus, the kernel of *self* can be split in three parts: E_f, a second part A, and $[\texttt{cnst}(\alpha_{aux}(map(\kappa_e(|E_f :: A|))(E_f :: A)))]$, 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 with 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, part E_f uses the Gödel number of *self* accordingly to the programmer's desire.
The function α_{aux} was recursively specified from the back to the front to be adapted to $\mathit{self}'4$ (in which a natural, related to the first element, is calculated before another natural, related to the last element, is calculated). This decision reduces the effort necessary in the formalization since it allows for avoiding the elaborated analysis that a recursive specification from the front to the back would imply. In such an alternative version of $\mathit{self}'4$ the last element should represent a stack of naturals associated with each element in E_f :: A by the function κ_e. In such a case, to calculate the Gödel number of the alternative version of *self* it would be necessary to add the number associated with its last element to the bottom of the stack.
The second part of *self*, A, is defined as below, where δ is the greatest index of `rec` found in the list E_f, using the function *printA*:

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

The function *printA* is specified below.

$$\begin{aligned}
&\mathit{printA}(\mathit{len}, \mathit{len}2) := \\
&[\kappa_2^S(\texttt{cnst}(1 + \mathit{len} + \mathit{len}2 + |\mathit{mult}|), \kappa_2^S(\texttt{rec}(|\mathit{mult}| + \mathit{len} + 1, \texttt{vr}), \\
&\mathit{succ}^S(\texttt{rec}(1, \kappa_2^S(\texttt{cnst}(5), \texttt{rec}(|\mathit{mult}| + \mathit{len} + 1, \texttt{vr}))))))] :: \\
&\mathit{mult}^{+1} :: [\texttt{vr}]^{\mathit{len}}
\end{aligned}$$

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

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

Since in the specification of A above the arguments of *printA* are δ and $|E_f|$, it is warranted that the indices of `rec` in *self* are always valid. Thus, *self* is `partial_recursive` because the restriction on basic operator is maintained by construction.
The list *mult*, used in the specification of *printA*, multiplies using *sum* that adds pairs of naturals encoded as a unique natural by the function κ_2 as below.

$$
\begin{aligned}
\mathit{sum} := [&\mathtt{ite}(\pi_1^S(\mathtt{vr}),\\
&\qquad \mathit{succ}(\mathtt{rec}(0, \kappa_2^S(\mathtt{rec}(1, \pi_1^S(\mathtt{vr})), \pi_2^S(\mathtt{vr})))),\\
&\qquad \pi_2^S(\mathtt{vr}))]\\
::\ &\mathit{sub}1'4^{+1}
\end{aligned}
$$

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

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

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

Formalizing the correctness of *mult* and *sum* directly is possible but hard to execute because the semantic evaluation generates a large chain of existential quantifiers. To avoid this difficulty, the formalization of the equivalence between the `MF-PVS0` specifications of *mult* and *sum* and their associated PVS functions, mult_f and sum_f were obtained. Also, the correctness of the associated PVS functions was shown. Thus, the correctness of *mult* and *sum* are given as corollaries.

The next lemma shows the correctness of *printA*.

Lemma 10 (Correctness of *printA* - `print_correctness`).

$$
\begin{aligned}
&\forall (i, \mathit{len}, \mathit{len}2, h):\\
&\quad \gamma\langle \mathit{printA}(\mathit{len}, \mathit{len}2) :: [\mathtt{cnst}(h)]\rangle\\
&\qquad\qquad (i, \kappa_2(1 + \mathit{len} + \mathit{len}2 + |\mathit{mult}|, \kappa_2(h, 5 \times h + 1)))
\end{aligned}
$$

To use this lemma, δ, $|E_f|$ and $\alpha_{aux}(\mathit{map}(\kappa_e(|E_f :: A|))(E_f :: A))$ are used to instantiate the variables *len*, *len*2 and h, respectively. In addition, $\mathit{self}'4 := E_f :: A :: [\mathtt{cnst}(h)]$. This gives:

$$
\begin{aligned}
&\forall (i):\\
&\quad \gamma\langle \mathit{printA}(\delta, |E_f|) :: [\mathtt{cnst}(h)]\rangle\\
&\qquad\qquad (i, \kappa_2(|\mathit{self}'4| - 1, \kappa_2(h, \kappa_e(|\mathit{self}'4| - 1)(\mathtt{cnst}(h)))))
\end{aligned}
$$

because

$$1 + \delta + |E_f| + |\mathit{mult}| = |\mathit{self}'4| - 1$$

$$5 \times h + 1 = \kappa_e(|\mathit{self}'4| - 1)(\mathtt{cnst}(h))$$

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

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

The result of this replacement is:

$$\begin{array}{l}
\forall(i) : \\
\quad \gamma\langle \mathit{printA}(\delta, |E_f|) :: [\mathtt{cnst}(h)]\rangle \\
\qquad (i, \kappa_2(|\mathit{self}'4|-1, \alpha_{aux}(\mathit{map}(\kappa_e(|\mathit{self}'4|-1))(\mathit{self}'4))))
\end{array}$$

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

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

Thus, it can be concluded that:

$$\begin{array}{l}
\forall(i) : \\
\gamma(\langle \mathit{printA}(\delta, |E_f|) :: [\mathtt{cnst}(h)]\rangle)(i, \kappa_p(\mathit{self}))
\end{array}$$

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

As discussed before Theorem 4, instead of using composition, minimization, and primitive recurrence operators implemented for the proof of Turing Completeness in the previous Section, the formalization approach is based on the direct implementation of `MF-PVS0` programs. This decision allows for the analysis of computational properties directly over the `MF-PVS0` model, avoiding using the theory of partial recursive functions. Of course, the composition, minimization and primitive recurrence operators may be applied for constructing `MF-PVS0` programs such as *sum*, *mult* and others (used in the formalization of Recursion Theorem in Section 4.2). Notice that this kind of construction makes the semantics of the programs difficult to understand. For instance, an alternative implementation of the program *sum* given in Section 4.2, can be done using composition and primitive recurrence as below.

$$\begin{array}{c}
\mathit{sum} := \mathit{prim\_recur}(\mathit{comp}([\mathit{succ}(\mathtt{vr})], [[\mathit{comp}(\mathit{proj}'4, \kappa_2^S(0, \kappa_2^S(2, \mathtt{vr})))'4]])'4, \\
\mathit{comp}(\mathit{proj}'4, [[\kappa_2^S(0, \kappa_2^S(0, \mathtt{vr}))]])'4)
\end{array}$$

The construction of the specialized Gödelization functions required to build *self* is the most difficult part of this formalization. One challenge in the implementation of κ_p was to build it in such a manner that it facilitates further steps of the formalization. An appropriate function α_{aux} was enough to reach this aim. Specifically for the Recursion Theorem, κ_p need not to be bijective. However, it was done in this way in order to make it useful for the formalization of other theorems such as the Fixed-Point Theorem. Ensuring that κ_p is bijective was technically difficult since it requires that all necessary auxiliary functions were also bijective. For some auxiliary components, the formalization was straightforward. However, for other ones, PVS infers some types such that applying some lemmas about lists (of `MF-PVS0` expressions, i.e., a kernel of `MF-PVS0` programs, and naturals) do not work. Such types appear when specific kernels of `MF-PVS0` programs were considered, such as those that included only valid recursive call indices. An example of such properties on lists is $|A :: B| = |A| + |B|$. There is an appropriate lemma for this property, but it requires that A, B, and A::B all have the same type. Nevertheless, if the type of A and B, say S, is a subtype of the type of $A :: B$, say T, being all the inputs of the function length ($|\_| \ : \ T \to \mathbb{N}]$), the lemma needs to be specialized and proved separately. The general solution, non-provided in PVS, is to prove that if S is a subtype of T, and $A \ : \ S$, then $|A|[T] = |A|[S]$. Indeed, in PVS proofs, the concatenation of lists of naturals is interpreted as lists of numbers. Similar type inference problems were encountered when formalizing the Recursion Theorem

Chapter 5

Formalization of the computational properties of the `PVS0` Model - Rice's Theorem, and Fixed Point Theorem

This Chapter continues the discussion on the technicalities of the PVS formalizations of computational properties of the `PVS0` model, which started in the previous chapter. Section 5.1 describes the formalization of Rice's Theorem obtained as a corollary of the Recursion Theorem. Furthermore, it discusses other corollaries derived from Rice's Theorem. Finally, Section 5.2 describes the formalization of the Fixed Point Theorem. For the last formalization, it was necessary to transform the Gödel number on the correspondent program and another program in a Gödel number to formalize it.

5.1 Rice's Theorem

Rice's Theorem is a consequence of the Recursion Theorem. It is proved that if using the basic built-in operators, the Recursion Theorem holds then Rice's Theorem for this theory also holds. Notice the basic operators used in theory `mf_pvs0_Recursion_Theorem` to guarantee this theorem, and those used in theory `mf_pvs0_Turing_Completeness` to ensure Turing Completeness are the same. The formalization in this section proves that for all Gödelizations for which the Recursion Theorem holds, and Rice's Theorem also holds. Similarly to standard demonstrations of the undecidability of the Halting Problem and the uncountability of real numbers, the formalization is based on Cantor's diagonal argument. An alternative formalization approach is based on the construction of a universal program for the `MF-PVS0` model. However, this approach would increase the complexity of the formalization. Using such a proof strategy, the formalization would require the construction of an elaborated reduction of the Halting Problem to the problem of separability of extensional properties of MF-PVS0 programs. Thus, the current formalization does not use (undecidability of) the Halting Problem and depends only on the above-mentioned Theorem 4.

Formalizing Rice's Theorem also requires a definition of *extensional property* of programs.

The notion of an extensional property over MF-PVS0 programs is specified as:

$$\begin{aligned} &\mathit{is\_semantic\_predicate?}(P) := \forall(\mathit{pvs_0}_1, \mathit{pvs_0}_2) : \\ &(\forall(v_i, v_o) : \gamma(\mathit{pvs_0}_1)(v_i, v_o) \Leftrightarrow \gamma(\mathit{pvs_0}_2)(v_i, v_o)) \Rightarrow \\ &\qquad\qquad\qquad (P(\mathit{pvs_0}_1) \Leftrightarrow P(\mathit{pvs_0}_2)) \end{aligned} \tag{5.1}$$

If *is_semantic_predicate?* holds for the predicate P, it is said that P is an extensional property. Extensional property means that it must hold for programs that perform the same and it must not hold for programs that do not perform the same. For example, performing a greater common divisor is an extensional property. However, verifying if a program has less than ten lines of code is not an extensional property.

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

Theorem 5 (Rice's Theorem - `Rice_theorem_for_Turing_complete_pvs0`).

$$\forall(P : \texttt{is\_semantic\_predicate}) : \left(\begin{array}{l} \exists(\mathit{decider} : \texttt{computable}) : \\ \quad \forall(\mathit{pvs_0} : \texttt{partial\_recursive}) : \\ \qquad (\neg\gamma(\mathit{decider})(\kappa_p(\mathit{pvs_0}), 0)) \Leftrightarrow P(\mathit{pvs_0}) \end{array} \right) \Leftrightarrow \quad (P = \mathit{fullset} \lor P = \emptyset)$$

Necessity: Suppose that $P = \mathit{fullset}$. Let $\top$ be an element different from 0. The MF-PVS0 program $\mathit{decider} = \langle[\texttt{cnst}(\top)]\rangle$, decides *fullset*. Now, suppose that $P = \emptyset$. The MF-PVS0 program $\mathit{decider} = \langle[\texttt{cnst}(0)]\rangle$ decides $\emptyset$.
Sufficiency: by contraposition, let assume that $(P \neq \mathit{fullset} \land P \neq \emptyset)$. This implies that there exist MF-PVS0 programs, say p and np, such that $P(p)$ and $\neg P(np)$.
For reaching a contradiction, suppose that there exists *decider* : `computable` such that:

$$\forall(\mathit{pvs_0} : \texttt{partial\_recursive}) : \; \neg\gamma(\mathit{decider})(\kappa_p(\mathit{pvs_0}), 0) \Leftrightarrow P(\mathit{pvs_0})$$

And, consider the program *opp* with the kernel:

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

Using the Recursion Theorem (Theorem 4) that there are programs in the model that can print their own Gödel number, making $E_f = \mathit{opp}$:

$$\begin{aligned}
& \exists(\mathit{print} : \mathtt{partial\_recursive}) :\\
& \quad \mathtt{LET} \quad \mathit{self} = \langle \mathit{opp} :: \mathit{print}'4^{+|\mathit{opp}|} \rangle \ \ \mathtt{IN}\\
& \quad\quad \mathit{partial\_recursive?}(\mathit{self}) \ \wedge\\
& \quad\quad \forall (i) \ : \varepsilon(\mathit{self})(\mathit{print}'4(0)^{+|\mathit{opp}|}, i, \kappa_p(\mathit{self}))
\end{aligned}$$

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

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

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

The Recursion Theorem 4 implies that there exists an element of the partial recursive class, say *print*, such that:

$$\begin{aligned}
& \mathtt{LET} \quad \mathit{self} = \langle \mathit{opp} :: \mathit{print}'4^{+|\mathit{opp}|} \rangle \ \ \mathtt{IN}\\
& \quad \mathit{partial\_recursive?}(\mathit{self}) \ \wedge\\
& \quad \forall (i) \ : \ \varepsilon(\mathit{self})(\mathit{print}'4(0)^{+|\mathit{opp}|}, i, \kappa_p(\mathit{self}))
\end{aligned}$$

Making $\mathit{pvs}_0 = \mathit{self}$ it can be concluded that

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

The proof splits into two sub-cases.

Sub-case 1: $\boldsymbol{P(\mathit{self})}$. In this case, $\neg\gamma(\mathit{decider})(\kappa_p(\mathit{self}), 0)$ holds. Therefore, the program *self*, by definition, performs as *np*. But P is an extensional property, which means

that if two programs perform the same, then P holds for both or does not. Thus, $P(\mathit{self})$ and $P(np)$ must hold, that is a contradiction, because $\neg P(np)$.
Since P is an extensional property, one has:

$$\begin{aligned}&\forall(pvs_{O1}, pvs_{O2}) :\\ &(\forall(i,o) : \gamma(pvs_{O1})(i,o) \Leftrightarrow \gamma(pvs_{O2})(i,o)) \Rightarrow\\ &\qquad (P(pvs_{O1}) \Leftrightarrow P(pvs_{O2}))\end{aligned}$$

Thus, choosing pvs_{O1} as self and pvs_{O2} as np, it gives:

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

Assuming $\forall(i,o) : \gamma(\mathit{self})(i,o) \Leftrightarrow \gamma(np)(i,o)$, by $P(\mathit{self})$, $P(np)$ also holds, which is a contradiction since $\neg P(np)$.
Consequently, $\neg\forall(i,o) : \gamma(\mathit{self})(i,o) \Leftrightarrow \gamma(np)(i,o)$ should hold.
But this is not possible because *self* performs the same as *np* as shown below.
Starting by $\gamma(\mathit{self})(i,o)$ and expanding γ, and from $\varepsilon(\mathit{self})(\mathit{self}'4(0), i, o)$ replacing *self* by its definition, one obtains:

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

By properties of lists and definition of *opp* gives $\varepsilon(\mathit{self})(opp(0), i, o)$. Therefore:

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

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

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

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

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

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

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

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

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

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

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

Then, by using the shift code lemma (Lemma 1) one obtains the equivalence below.

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

Thus, one obtains,

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

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

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

By adequate expansions of predicate ε, Skolemization of the obtained existentially quantified variable as v_1' and replacing the necessary variables, one obtains:

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

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

Sub-case 2: $\boldsymbol{\neg P(\mathit{self})}$. It follows analogously to sub-case 1, except that *self* performs as p. Thus, $\neg P(\mathit{self})$ and $\neg P(p)$ hold. It gives a contradiction because $P(p)$ holds. Thus, there exists no `MF-PVS0` program that decides any extensional property different from the total or empty one.

Next, details of the formalization are included.

First, notice that the formula below holds:

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

In this case, $\gamma(\mathit{decider})(\kappa_p(\mathit{self}), 0)$ is concluded.

P is an extensional property:

$$\forall(pvs_{0\,1}, pvs_{0\,2}) : (\forall(i, o_1, o_2) : \gamma(pvs_{0\,1})(i, o_1) \wedge \gamma(pvs_{0\,2})(i, o_2) \Rightarrow o_1 = o_2) \\ \Rightarrow (P(pvs_{0\,1}) \Leftrightarrow P(pvs_{0\,2}))$$

Choosing $pvs_{0\,1}$ as *self* and $pvs_{0\,2}$ as p:

$$(\forall(i, o_1, o_2) : \gamma(\mathit{self})(i, o_1) \wedge \gamma(p)(i, o_2) \Rightarrow o_1 = o_2) \Rightarrow (P(\mathit{self}) \Leftrightarrow P(p))$$

Supposing the premise of this implication, one has that $P(\mathit{self}) \Leftrightarrow P(p)$, but in this case $\neg P(\mathit{self})$, and therefore $\neg P(p)$, which is a contradiction since $P(p)$.

Thus, $\neg\forall(i, o_1, o_2) : \gamma(\mathit{self})(i, o_1) \wedge \gamma(p)(i, o_2) \Rightarrow o_1 = o_2$ holds, which is equivalent to $\exists(i, o_1, o_2) : \gamma(\mathit{self})(i, o_1) \wedge \gamma(p)(i, o_2) \wedge o_1 \neq o_2$.

By Skolemization of i, o_1 and o_2, using the same variable names, and expanding γ definition:

$$\varepsilon(\mathit{self})(\mathit{self}'4(0), i, o_1) \quad\wedge\quad \varepsilon(p)(p'4(0), i, o_2) \quad\wedge\quad o_1 \neq o_2$$

From $\varepsilon(\mathit{self})(\mathit{self}'4(0), i, o_1)$, by replacing the second occurrence of *self* by its definition and simplifying one obtains:

$$\varepsilon(\mathit{self})((opp :: \mathit{print}'4^{|opp|})(0), i, o_1)$$

By properties of lists:

$$\varepsilon(\mathit{self})(opp(0), i, o_1)$$

Expanding *opp* definition and simplifying the access of the first element of the list one obtains:

$$\begin{aligned}\varepsilon(\mathit{self})(\quad & \mathtt{ite}(\mathtt{rec}(1, \mathtt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4| + |p'4|, \mathtt{vr})),\\ & \quad \mathtt{rec}(1 + |\mathit{decider}'4|, \mathtt{vr}),\\ & \quad \mathtt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \mathtt{vr}), \qquad\qquad i, o_1)\end{aligned}$$

By the definition of ε and the operational semantics of `ite`:

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

Expanding the first occurrence of ε:

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

Expanding the first occurrence of ε again:

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

Expanding the first occurrence of ε again:

$$\begin{aligned}&\exists\,(v') : \exists\,(v'') : \exists\,(v''') : i = v''' \ \wedge\\ &\varepsilon(\mathit{self})(\mathit{self}'4(1 + |\mathit{decider}'4| + |\mathit{np}'4| + |p'4|), v''', v'') \ \wedge\\ &\varepsilon(\mathit{self})(\mathit{self}'4(1), v'', v') \ \wedge\\ &\mathtt{IF}\ v' \neq \bot\ \mathtt{THEN}\ \varepsilon(\mathit{self})(\mathtt{rec}(1 + |\mathit{decider}'4|, \mathtt{vr}), i, o_1)\\ &\mathtt{ELSE}\ \varepsilon(\mathit{self})(\mathtt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \mathtt{vr}), i, o_1)\end{aligned}$$

Applying the equalities $\mathit{self}'4(1) = \mathit{decider}'4(0)^{+1}$, $\mathit{self}'4(1 + |\mathit{decider}'4| + |\mathit{np}'4| + |p'4|) = \mathit{print}'4(0)^{+|\mathit{opp}|}$:

$$\begin{aligned}&\exists\,(v') : \exists\,(v'') : \exists\,(v''') : i = v''' \ \wedge\\ &\varepsilon(\mathit{self})(\mathit{print}'4(0)^{+|\mathit{opp}|}, v''', v'') \ \wedge\\ &\varepsilon(\mathit{self})(\mathit{decider}'4(0)^{+1}, v'', v') \ \wedge\\ &\mathtt{IF}\ v' \neq \bot\ \mathtt{THEN}\ \varepsilon(\mathit{self})(\mathtt{rec}(1 + |\mathit{decider}'4|, \mathtt{vr}), i, o_1)\\ &\mathtt{ELSE}\ \varepsilon(\mathit{self})(\mathtt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \mathtt{vr}), i, o_1)\end{aligned}$$

By Skolemization of the existentially quantified variable:

$$\begin{array}{l} i = v''' \ \wedge \\ \varepsilon(\mathit{self})(\mathit{print}'4(0)^{+|\mathit{opp}|}, v''', v'') \ \wedge \\ \varepsilon(\mathit{self})(\mathit{decider}'4(0)^{+1}, v'', v') \ \wedge \\ \texttt{IF}\ v' \neq \bot\ \texttt{THEN}\ \varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4|, \texttt{vr}), i, o_1) \\ \texttt{ELSE}\ \varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \texttt{vr}), i, o_1) \end{array}$$

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

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

The relation ε is functional. It means that, $v'' = \kappa_p(\mathit{self})$.

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

Using the lemmas of code shift,
$\varepsilon(\mathit{self})(\mathit{decider}'4(0)^{+1}, \kappa_p(\mathit{self}), v') \Leftrightarrow \varepsilon(\mathit{decider})(\mathit{decider}'4(0), \kappa_p(\mathit{self}), v')$.
Thus,

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

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

$$\varepsilon(\mathit{self})(\texttt{rec}(1 + |\mathit{decider}'4| + |\mathit{np}'4|, \texttt{vr}), i, o_1)$$

Expanding ε:

$$\exists\,(v') : \varepsilon(\mathit{self})(\texttt{vr}, i, v') \ \wedge \varepsilon(\mathit{self})(\mathit{self}'4(1 + |\mathit{decider}'4| + |\mathit{np}'4|), v', o_1)$$

Then, expanding the first occurrence of ε:

$$\exists\,(v') : i = v' \ \wedge \varepsilon(\mathit{self})(\mathit{self}'4(1 + |\mathit{decider}'4| + |\mathit{np}'4|), v', o_1)$$

By Skolemizing and Replacing v' by i:

$$\varepsilon(\mathit{self})(\mathit{self}'4(1 + |\mathit{decider}'4| + |\mathit{np}'4|), i, o_1)$$

Replacing $\mathit{self}'4(1 + |\mathit{decider}'4| + |\mathit{np}'4|)$ by $p'4(0)$

$$\varepsilon(\mathit{self})(p'4(0), i, o_1)$$

Applying lemmas of code shift:

$$\varepsilon(\mathit{np})(p'4(0), i, o_1)$$

From the first formula at the beginning of this sub-case, $\varepsilon(p)(p'4(0), i, o_2) \wedge o_1 \neq o_2$:

$$\varepsilon(p)(p'4(0), i, o_1) \wedge \varepsilon(p)(p'4(0), i, o_2) \wedge o_1 \neq o_2$$

Since ε is a functional relation, $o_1 = o_2$ holds, which gives a contradiction.
Thus, there exists no MF-PVS0 program *decide* that decides an extensional property different from the total or empty one. □

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

Corollary 1 (Undecidability of the Uniform Halting Problem - `uniform_halting_problem_undecidability_Turing_complete`).

$$\begin{array}{l} \neg\exists(\mathit{decider} : \texttt{computable}) : \\ \quad \forall(\mathit{pvs_0} : \texttt{partial\_recursive}) : \\ \quad\quad (\neg\gamma(\mathit{decider})(\kappa_p(\mathit{pvs_0}), 0) \Leftrightarrow T_\varepsilon(\mathit{pvs_0})) \end{array}$$

The formalization uses Rice's Theorem instantiating the extensional property as T_ε. The predicate T_ε is an extensional property because if two MF-PVS0 programs perform the same, both are either terminating or not. Since the set T_ε is neither equal to the empty set nor the whole set `partial_recursive`, there exists no computable decider for this set. To prove this, it is shown that the `partial_recursive` constant program $\langle[\texttt{cnst}(0)]\rangle$ belongs to T_ε, while a simple loop `partial_recursive` program specified as $\langle[\texttt{rec}(0, \texttt{vr})]\rangle$ does not.
For the loop above, notice that the input of the recursive call does not change. Therefore, the execution of the program will repeat the recursive call infinitely.

The PVS theory complementing this thesis includes both the formalization of the corollary above and a direct formalization of the undecidability of the (Specific) Halting

Problem for the multiple-function MF-PVS0 model in the spirit of (Ferreira Ramos et al. [2018]).

Corollary 2 (Undecidability of Existence of Fixed Points - `fixed_point_existence_undecidability_Turing_complete`).

$$\begin{array}{l}\neg\exists(\mathit{decider} : \texttt{computable}) :\\ \quad \forall(\mathit{pvs_0} : \texttt{partial\_recursive}) :\\ \quad\quad (\neg\gamma(\mathit{decider})(\kappa_p(\mathit{pvs_0}), 0) \Leftrightarrow \exists(p) : \gamma(\mathit{pvs_0})(p, p))\end{array}$$

The formalization instantiates Rice's Theorem using the extensional property

$$\lambda(\mathit{pvs_0} : \texttt{partial\_recursive}) : \exists(p) : \gamma(\mathit{pvs_0})(p, p)$$

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

Corollary 3 (Undecidability of Self Replication - `self_replication_ undecidability_Turing_complete`).

$$\begin{array}{l}\neg\exists(\mathit{decider} : \texttt{computable}) :\\ \quad \forall(\mathit{pvs_0} : \texttt{partial\_recursive}) :\\ \quad\quad (\neg\gamma(\mathit{decider})(\kappa_p(\mathit{pvs_0}), 0) \Leftrightarrow\\ \quad\quad\quad\quad \exists(p : \texttt{partial\_recursive}) :\\ \quad\quad\quad\quad\quad \forall(i) : \gamma(p)(v_i, \kappa_p(p)) \wedge \gamma(\mathit{pvs_0})(v_i, \kappa_p(p)))\end{array}$$

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

$$\begin{array}{l}\lambda(\mathit{pvs_0} : \texttt{partial\_recursive}) :\\ \quad \exists(p : \texttt{partial\_recursive}) :\\ \quad\quad \forall(i) : \gamma(p)(i, \kappa_p(p)) \wedge \gamma(\mathit{pvs_0})(i, \kappa_p(p))\end{array}$$

The predicate above is an extensional property because if two MF-PVS0 programs behave in the same way, either both return a Gödel number of a program that self-replicates or neither do.

The next step consists in showing that the predicate is neither the empty set nor the full `partial_recursive` set. Using the assumption of the Recursion Theorem and instantiating it with $[\texttt{rec}(1, \texttt{vr})]$, one shows that the predicate is not empty. On the other side, the program $\langle[\texttt{op2}(i, \texttt{cnst}(1), \texttt{vr})]\rangle$ shows that the predicate is not the whole `partial_recursive` set.

Corollary 4 (Undecidability of Functional Equivalence - `pvs0_program_ equivalence_undecidability_Turing_complete`).

$$
\begin{aligned}
&\neg\exists(\mathit{decider} : \texttt{computable}) : \\
&\quad \forall(\mathit{pvs}_{0\,0}, \mathit{pvs}_{0\,1} : \texttt{partial\_recursive}) : \\
&\qquad (\neg\gamma(\mathit{decider})(\kappa_2(\kappa_p(\mathit{pvs}_{0\,0}), \kappa_p(\mathit{pvs}_{0\,1})), 0) \Leftrightarrow \\
&\qquad\quad (\ \forall(v_i, v_o) : \gamma(\mathit{pvs}_{0\,0})(v_i, v_o) \Leftrightarrow \gamma(\mathit{pvs}_{0\,1})(v_i, v_o)\ )\ \ )
\end{aligned}
$$

Suppose that there exists a `computable` program *decider* that decides the above equivalence between `MF-PVS0` `partial_recursive` programs. Then, instantiate $\mathit{pvs}_{0\,0}$ above as the constant zero program, $\langle[\texttt{cnst}(0)]\rangle$ simplifying in this manner the problem to decide whether a program performs the same as the constant zero program. The next step is instantiating Rice's Theorem (Theorem 5) with the predicate below.

$$
\begin{aligned}
&\lambda(\mathit{pvs}_0 : \texttt{partial\_recursive}) : \\
&\quad \forall(v_i, v_o) : \\
&\qquad \gamma\langle[\texttt{cnst}(0)]\rangle(v_i, v_o) \Leftrightarrow \gamma(\mathit{pvs}_0)(v_i, v_o)
\end{aligned}
$$

Indeed, the predicate above is an extensional property because either two `MF-PVS0` programs always return zero or not.

To prove that the predicate neither is empty nor the full `partial_recursive` set, it is enough to show that the constant zero and one programs respectively belong and do not to the predicate. After that, one uses the assumed program *decider* to build another program for deciding the equivalence to the constant zero program; this program is built as $\langle[\texttt{rec}(1, \texttt{op2}(i, \kappa_p\langle[\texttt{cnst}(0)]\rangle), vr))] :: \mathit{decider}`4\rangle$, where i is the index of the κ_2 function. Indeed, by using the shifting code lemmas, this program can be adapted to decide the equivalence with the constant zero program.

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

Corollary 5 (Undecidability of Natural Predicate - `natural_predicate_undecidability` ↗).

$$\begin{aligned}&\neg\forall(P : \mathbb{N} \to \mathit{Boolean}) : \exists(\mathit{decider} : \texttt{computable}) :\\ &\quad \forall(i : \mathbb{N}) :\\ &\qquad (\neg\gamma(\mathit{decider})(i, 0)) \Leftrightarrow P(i)\end{aligned}$$

Suppose that all predicates of the natural numbers have a *decider* that decides if this predicate holds for a number. Instantiate the predicate as:

$$P(i) = \exists(p) : \gamma(\kappa_p^{-1}(i))(p, p))$$

This is a contradiction because *decider* is deciding the existence of the fixed point in a program represented by the Gödel number i.

5.2 Fixed Point Theorem

The Fixed Point Theorem (Rogers' Fixed Point Theorem) and the undecidability of fixed point existence are distinct properties. The undecidability principle states that determining whether a program has a fixed point result is impossible. On the other hand, the Fixed Point Theorem asserts that for any program f, a program p exists such that the execution of $f(p)$ produces the same outcome as executing p.

In addition, Rogers' Fixed Point Theorem can be applied when designing algorithms for computing optimal solutions. It provides insight into approaching these problems mathematically so they may be solved computationally efficiently. For instance, this theorem allows us to determine if a given algorithm will converge towards a solution quickly enough without having too much associated computational cost during execution time; this helps ensure efficient use of resources while still achieving desired results in reasonable amounts of time (Istratescu [1981]).

Rogers' Fixed Point Theorem has been used extensively within computational logic and artificial intelligence because it provides insights into how complex systems interact. In particular, this theorem has been employed to develop intelligent agents capable of learning from the environment around them via reinforcement techniques such as Q-learning and SARSA algorithms (Mnih et al. [2015]). As such, Rogers' Fixed Point Theorem is a valuable tool in helping computer scientists design better systems more effectively and solve real-world problems efficiently through the computation power available in today's technology landscape.

In the case of partial recursive `MF-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 `MF-PVS0` program. Using these operators it also must be possible to build a `MF-PVS0` program such that it receives a natural as an argument, and split it into another two arguments, a and b. The natural a is a Gödel number of a `MF-PVS0` program applied to the own a, resulting in another Gödel number of another program applied to b. This last `MF-PVS0` program is called *diagonal*.

The formalization consists in building the `MF-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.

Thus, to formalize the Fixed Point Theorem, it is required three assumptions, over arbitrary built-in operators, that are parameters of the theory ↗.

The first assumption says that there is a universal `MF-PVS0` program.

$$\begin{aligned}&\exists(universal : \texttt{computable}) : \forall(m, n, o) :\\&\gamma(universal)(\kappa_2(m, n), o) \Leftrightarrow \gamma(\kappa_p^{-1}(m))(n, o)\end{aligned}$$

The second assumption says that one of the binary operators is the codification of a tuple of naturals in naturals.

$$\exists(k < |O_2|) : O_2(k) = \kappa_2$$

The third assumption is:

$$\begin{aligned}&\forall(j < |O_2|, u) : \exists(diagonal : \texttt{computable}) : \forall(i) :\\&\texttt{LET}\ m = \langle O_1, O_2, \bot, [\texttt{rec}(1, \texttt{op2}(j, \texttt{rec}(1, \texttt{op2}(j, \texttt{cnst}(i), \texttt{cnst}(i)), \texttt{vr})))] :: u^{+1}\rangle \texttt{IN}\\&partial\_recursive(m) \wedge \gamma(diagonal)(i, \kappa_p(m))\end{aligned}$$

The idea under this last assumption is replacing j with the index of the operator κ_2 given by the second assumption and replacing u with the fourth element of the universal `MF-PVS0` program (a list of the `MF-PVS0` expressions) given by the first assumption to extract the diagonal `MF-PVS0` program. This program models the lambda term $\lambda a, e{\cdot}(aa)e$.

It is also necessary to define a function that applies for a `computable` `MF-PVS0` program in a `partial_recursive` `MF-PVS0` program. Remark that the `MF-PVS0` programs here have as input and output the naturals. Thus to make this application is necessary, firstly, Gödelizing the `partial_recursive` `MF-PVS0` program input, transforming it in a natural, secondly, applying the `computable` `MF-PVS0` program to the natural and, finally, taking the output to degödelizing it, transforming the natural number result in another `partial_recursive` program. The function is defined below.

$$\Delta(comp)(part) := \kappa_p^{-1}(choose\{o : \mathbb{N} \mid \gamma(comp)(\kappa_p(part), o)\})$$

Theorem 6. Fixed-Point Theorem for `MF-PVS0` - `fixed_point`

$$\forall(f : \texttt{computable}) : \exists(p : \texttt{partial\_recursive}) :$$
$$\forall(i, o_1, o_2) : \gamma(p)(i, o_1) \wedge \gamma(\Delta(f)(p))(i, o_2) \Rightarrow o_1 = o_2$$

The idea of the proof follows the same lines as the demonstration of the Fixed Point theorem for Turing machines presented in (Floyd and Beigel [1994]). The approach is adapted for lambda calculus as below.

$$\forall(F) : \exists(P) : FP =_{\beta\eta} P$$

Let F be any lambda term. Let $D := \lambda a, e \cdot (aa)e$. Let $P = D(\lambda z \cdot (F(Dz)))$.

$$\begin{gathered} P = \\ D(\lambda z \cdot (F(Dz))) =_\beta \\ (\lambda a, e \cdot (aa)e)(\lambda z \cdot (F(Dz))) =_\beta \\ (\lambda e \cdot ((\lambda z \cdot (F(Dz)))(\lambda z \cdot (F(Dz))))e) =_\beta \\ (\lambda e \cdot ((F(D(\lambda z \cdot (F(Dz))))))e) = \\ (\lambda e \cdot ((FP))e) =_\eta \\ FP \end{gathered}$$

The formalization starts considering $universal$ as the universal program getting from the first assumption, k as the index of the binary operator such that it returns the the function κ_2 according to the second assumption and $diagonal$ as the program such that according to the third assumption choosing j as k and u as $universal'4$.

Let f be any `computable` program. Let $comp := \langle O_1, O_2, \bot, [\texttt{rec}(1, \texttt{rec}(1 + |f'4|))] :: f'4^{+1} :: diagonal'4^{+1+|f'4|}\rangle$. In this case, $comp$ is the composition of f and the $diagonal$.

Taking $p := \Delta(diagonal)(comp)$.

Expanding γ in $\gamma(p)(i, o_1)$, after expanding ε, applying the shifting code lemmas, it is possible to note that p produces the same output of $\Delta(f)(p)$ for the same input.

Starting with the hypothesis :

$$\gamma(p)(i, o_1)$$

Replacing p with this definition:

$$\gamma(\Delta(diagonal)(comp))(i, o_1)$$

Expanding Δ :

$$\gamma(\kappa_p^{-1}(choose\{o : \mathbb{N} \mid \gamma(diagonal)(\kappa_p(comp), o)\}))(i, o_1) \tag{5.2}$$

Let $h = choose(\{o : \mathbb{N} \mid \gamma(diagonal)(\kappa_p(comp), o)\})$

Thus, it holds:

$$\gamma(diagonal)(\kappa_p(comp), h)$$

By the third assumption and γ being deterministic:

$$\kappa_p^{-1}(h) = m = \langle O_1, O_2, \bot, [$$
$$\texttt{rec}(1, \texttt{op2}(k, \texttt{rec}(1, \texttt{op2}(k, \texttt{cnst}(\kappa_p(comp)), \texttt{cnst}(\kappa_p(comp)))), \texttt{vr}))] ::$$
$$universal'4^{+1}\rangle$$

Replacing it in the proposition 5.2:

$$\gamma(m)(i, o_1)$$

Expanding γ:

$$\varepsilon(m)(\texttt{rec}(1, \texttt{op2}(k, \texttt{rec}(1, \texttt{op2}(k, \texttt{cnst}(\kappa_p(comp)), \texttt{cnst}(\kappa_p(comp)))), \texttt{vr})), i, o_1)$$

Expanding ε:

$$\exists\ v_1 :$$
$$\varepsilon(m)(\texttt{op2}(k, \texttt{rec}(1, \texttt{op2}(k, \texttt{cnst}(\kappa_p(comp)), \texttt{cnst}(\kappa_p(comp)))), \texttt{vr}), i, v_1) \wedge$$
$$\varepsilon(m)(universal'4(0)^{+1}, v_1, o_1)$$

Skolemizing v_1 and expanding the first ε :

$$\exists\ v_2 :$$
$$v_1 = \kappa_2(v_2, i) \wedge$$
$$\varepsilon(m)(\texttt{rec}(1, \texttt{op2}(k, \texttt{cnst}(\kappa_p(comp)), \texttt{cnst}(\kappa_p(comp)))), i, v_2) \wedge$$
$$\varepsilon(m)(universal'4(0)^{+1}, v_1, o_1)$$

Skolemizing v_2, replacing $v_1 = \kappa_2(v_2, i)$ and expanding the first ε :

$$\exists\ v_3 :$$
$$\varepsilon(m)(\texttt{op2}(k, \texttt{cnst}(\kappa_p(comp)), \texttt{cnst}(\kappa_p(comp))), i, v_3) \wedge$$
$$\varepsilon(m)(universal'4(0)^{+1}, v_3, v_2) \wedge \varepsilon(m)(universal'4(0)^{+1}, \kappa_2(v_2, i), o_1)$$

Skolemizing v_3 and expanding the first ε :

$$v_3 = \kappa_2(\kappa_p(comp), \kappa_p(comp))$$
$$\varepsilon(m)(universal'4(0)^{+1}, v_3, v_2) \wedge \varepsilon(m)(universal'4(0)^{+1}, \kappa_2(v_2, i), o_1)$$

Replacing $v_3 = \kappa_2(\kappa_p(comp), \kappa_p(comp))$:

$$\varepsilon(m)(universal'4(0)^{+1}, \kappa_2(\kappa_p(comp), \kappa_p(comp)), v_2)\wedge$$
$$\varepsilon(m)(universal'4(0)^{+1}, \kappa_2(v_2, i), o_1)$$

Applying shifting code lemmas:

$$\varepsilon(universal)(universal'4(0), \kappa_2(\kappa_p(comp), \kappa_p(comp)), v_2)\wedge$$
$$\varepsilon(universal)(universal'4(0), \kappa_2(v_2, i), o_1)$$

Applying the first assumption:

$$\varepsilon(comp)(comp'4(0), \kappa_p(comp), v_2) \wedge \varepsilon(\kappa_p^{-1}(v_2))(\kappa_p^{-1}(v_2)'4(0), i, o_1)$$

The program *comp* is the composition of f and the *diagonal*. This result comes expanding exhaustively the first ε and applying the shifting code lemmas:

$$\exists\ v_4 : \varepsilon(f)(f'4(0), v_4, v_2)\wedge$$
$$\varepsilon(diagonal)(diagonal'4(0), \kappa_p(comp), v_4)\wedge$$
$$\varepsilon(\kappa_p^{-1}(v_2))(\kappa_p^{-1\prime}4(v_2)(0), i, o_1)$$

Replacing ε by the γ definition and skolemizing v_4

$$\gamma(f)(v_4, v_2) \wedge \gamma(diagonal)(\kappa_p(comp), v_4) \wedge \gamma(\kappa_p^{-1}(v_2))(i, o_1)$$

By γ being deterministic, v_4 is a unique result of the second γ. Thus,

$$\gamma(f)(v_4, v_2)\wedge$$
$$v_4 = choose(o : \mathbb{N} \mid \gamma(diagonal)(\kappa_p(comp), o))\wedge$$
$$\gamma(\kappa_p^{-1}(v_2))(i, o_1)$$

Applying κ_p^{-1} in both sides of the equality:

$$\gamma(f)(v_4, v_2)\wedge$$
$$\kappa_p^{-1}(v_4) = \kappa_p^{-1}(choose(o : \mathbb{N} \mid \gamma(diagonal)(\kappa_p(comp), o)))\wedge$$
$$\gamma(\kappa_p^{-1}(v_2))(i, o_1)$$

Applying the Δ definition:

$$\gamma(f)(v_4, v_2) \wedge \kappa_p^{-1}(v_4) = \Delta(diagonal)(comp) \wedge \gamma(\kappa_p^{-1}(v_2))(i, o_1)$$

Replacing $p = \Delta(diagonal)(comp)$:

$$\gamma(f)(v_4, v_2) \wedge \kappa_p^{-1}(v_4) = p \wedge \gamma(\kappa_p^{-1}(v_2))(i, o_1)$$

The functions κ_p and κ_p^{-1} are inverse. In this point of the formalization it was realized that the function to Gödelize must be bijective, which is different from the formalizations of the undecidability of the Halting Problem and Rice's Theorem where it s only necessary to be injective:

$$\gamma(f)(\kappa_p(\kappa_p^{-1}(v_4)), v_2) \wedge \kappa_p^{-1}(v_4) = p \wedge \gamma(\kappa_p^{-1}(v_2))(i, o_1)$$

Replacing $\kappa_p^{-1}(v_4) = p$

$$\gamma(f)(\kappa_p(p), v_2) \wedge \gamma(\kappa_p^{-1}(v_2))(i, o_1)$$

By γ being deterministic, v_2 is unique:

$$v_2 = \mathit{choose}(\{o : \mathbb{N} \mid \gamma(f)(\kappa_p(p), o)\}) \wedge \gamma(\kappa_p^{-1}(v_2))(i, o_1)$$

Replacing $v_2 = \mathit{choose}(\{o : \mathbb{N} \mid \gamma(f)(\kappa_p(p), o)\})$

$$\gamma(\kappa_p^{-1}(\mathit{choose}(\{o : \mathbb{N} \mid \gamma(f)(\kappa_p(p), o))\}))(i, o_1)$$

Applying the definition of Δ:

$$\gamma(\Delta(f)(p))(i, o_1)$$

Now, suppose for any o_2:

$$\gamma(\Delta(f)(p))(i, o_2)$$

By the determinism of γ, $o_1 = o_2$ holds.

Thus, for all f `MF-PVSO` program, exists a fixed point p.

Chapter 6

Discussion on the formalization of the undecidability of other problems - Word Problem and the Post Correspondence Problem

After formalizing theorems about the PVS0 model, the next step is to prove theorems about other kinds of problems as consequences of the computational properties of the PVS0 model.

We selected two well-known problems: the undecidability of the Word Problem for Thue systems (WP) and the undecidability of the Post Correspondence Problem (PCP). In his seminal paper, Emil Post proved the undecidability of the Word Problem as a corollary of the undecidability of the Halting Problem for Turing Machines (Post [1947]).

The undecidability of the Halting Problem was first stated in a 1958 book (Davis [1958]) by Davis but it is attributed to Turing's paper (Turing [1937a]) according to (Lucas [2021])

Here, our objective is to discuss how to prove the undecidability of the WP as a consequence of the undecidability of the Halting Problem for the MF-PVS0 model. This requires elaborated additional work since transactions in Turing Machines are atomic operations that can be straightforwardly simulated as transactions on words over Thue systems. But the operations in the MF-PVS0 functional model are not atomic, which makes the simulation of the evaluation of MF-PVS0 programs through Thue systems harder than the simulation of transactions over Turing Machines. In particular, the simulation of recursion is not trivial, because each recursive call has different input to be represented by a word, and that must be linked to the variable symbol using rewriting rules of the Thue system. Since each recursive call potentially applies to an infinite number of different inputs, the main issue is to capture a set of infinite rules through a finite set of rules and symbols.

After obtaining the undecidability of WP, other problems can be proved undecidable by reductions from WP. In particular, we discuss how the undecidability of PCP can be proved as a consequence of the undecidability of WP as done in (Davis et al. [1994]). Also, it can be proved straightforward due to the Undecidability of the Halting Problem for Turing Machines (cf. Hopcroft et al. [2006]). The current formalization follows the reduction from WP to PCP approach. There is a myriad of undecidability results that can be derived from the undecidability of WP and PCP without referencing the `MF-PVS0` model, for instance, modified PCP (if an instance of PCP has a solution starting with a specific domino), the ambiguity of context-free grammars, if two context-free grammars do not have common words, among others (Sipser [2012]). Indeed, derivations of the undecidability of PCP formalized as part of the Coq library on undecidability problems include Binary PCP, Binary Stack Machine Termination Problem, Minsky Machines Termination Problem, Intuitionistic Linear Logic Provability, finding a general solution for a Diophantine equation (Hilbert's 10th Problem), undecidability of Higher-Order Unification Problem Larchey-Wendling and Forster [2022], Spies and Forster [2020].

Formalizing the properties related to WP and PCP requires some preliminary definitions given below. The proved results include the context-closedness of Thue Systems (lemma 11), the equivalence between congruence and parallel congruence in Thue Systems (lemma 12), lemmas about domino piece selection (lemmas 13, 14, and 15). Also, it is proved that having a congruence in a Thue system implies the existence of a related solution in the domino set obtained by translating the Thue system (lemma 16). In addition, it is proved that if a domino set is solvable, a minimal solution exists (lemma 17), and converse properties regarding the existence of congruences related to domino solutions (lemmas 16, and 18).

The lemma of reduction from the Halting Problem of `partial_recursive` to WP (lemma 19), and the theorem of Undecidability of WP (theorem 7) are discussed and specified but not fully formalized.

Let Σ be an alphabet. A list $E = [s_i \approx t_i | s_i, t_i \in \Sigma^*]$, for $i < n \in \mathbb{N}$, of rewrite rules is a Thue system. The empty word is denoted as $\square$. If two words u and v can be rewritten using equations from E, they are said to be congruent modulo E; this is denoted as $u \approx_E^? v$. Formally, $u \rightarrow_E v := \exists s_i, t_i, w, z : u = ws_iz \wedge v = wt_iz \wedge s_i \approx t_i \in E$. The converse of the relation $\rightarrow_E$ is denoted as $\leftarrow_E$ or $\rightarrow_E^\circ$. As usual, the transitive reflexive closure of a relation on words R is denoted as R^*. Then, $(\leftarrow_E \cup \rightarrow_E)^*$ is the reflexive transitive and symmetric closure of the relation $\rightarrow_E$, which, for short, is denoted as $\approx_E$. The relation $\rightarrow_E$ is called a reduction relation (from E). The following lemma states that

Thue systems are context closed.

Lemma 11 (Context closedness of Thue Systems).

$$\forall(s,t,w_1,w_2) : s \approx_E t \implies w_1 s w_2 \approx_E w_1 t w_2$$

The proof is by induction on the number of steps in the reflexive-transitive closure in
$$\forall(i,s,t,w_1,w_2) : s(\leftarrow_E \cup \rightarrow_E)^i t \implies w_1 s w_2 (\leftarrow_E \cup \rightarrow_E)^i w_1 t w_2$$
Induction basis The induction basis is for $i = 0$. In this case, $s = t$. Thus, $w_1 s w_2 = w_1 t w_2$, which implies $w_1 s w_2 \approx_E w_1 t w_2$.
Inductive step. Assume the theorem holds for $i = k$, and suppose $s(\leftarrow_E \cup \rightarrow_E)^{k+1} t$. This implies that there exists a word u such that $s(\leftarrow_E \cup \rightarrow_E)^k u$ and $u(\leftarrow_E \cup \rightarrow_E)t$. By induction hypothesis, $w_1 s w_2 (\leftarrow_E \cup \rightarrow_E)^k w_1 u w_2$ holds. Also, $w_1 u w_2 (\leftarrow_E \cup \rightarrow_E) w_1 t w_2$ holds by the definition of reduction. Thus, by transitivity, $w_1 s w_2 (\leftarrow_E \cup \rightarrow_E)^{k+1} w_1 t w_2$ holds.

Given a list of equations, E, the parallel reduction relation was also specified:

$$s \rightarrow_{||E} t := \exists m_1 \cdots m_{n+1} \bigwedge_{i=1}^{n} \exists s_i \approx t_i \in E :$$
$$s = m_1 s_1 \cdots m_n s_n m_{n+1} \land t = m_1 t_1 \cdots m_n t_n m_{n+1}$$

The specification uses lists to represent words as lists of symbols, sets of dominoes, and sequences of indices of dominoes that correspond to possible solutions. Thus, some basic operators on lists, available in the PVS prelude theory (or not), are applied. For instance, the recursive function that searches the position of an element e in a list l, for instance, a list of equations, is defined as:

```
search(e, l) :=  CASES l OF
                   []        :  0;
                   [h] :: t  :  IF h = e  THEN 0
                                ELSE  1 + search(e, t)
```

In order to build a set of dominoes corresponding to a Thue system, several auxiliary functions are required. First, the alphabet of the Thue system, Σ, will be extended with symbols $\overline{a}$ for each $a \in \Sigma$, and some other symbols that will be used to separate words and for determining which are the first and last played dominoes in a possible solution. Additionally, dominoes will be built for all rules, alphabet symbols, etc.

The first and second projections select the left-hand and right-hand sides of equations, respectively. For instance, $(s \approx t)'1 := s$, $(s \approx t)'2 := t$.

Reflexivity rules for the symbols of the alphabet are built by mapping the lambda function below over the list of alphabet symbols.

$$\alpha := map(\lambda a \cdot (a \approx a))(\Sigma)$$

Also, The converse of E is built by mapping the function below over the list of equations. Notice that the relation $\rightarrow_E^\circ = \rightarrow_{E^\circ}$.

$$E^\circ := map(\lambda e \cdot (e'2 \approx e'1))(E)$$

The lists of left and right-hand sides of a list of indices l of the set of equations E are concatenated using the functions S_1 and S_2, respectively, below.

$$S_1(E,l) := \begin{array}{ll} \texttt{CASES}\ l\ \texttt{OF} & \\ [] & : [] \\ [h] :: t & : E(h)'1 :: S_1(E,t) \end{array} \qquad S_2(E,l) := \begin{array}{ll} \texttt{CASES}\ l\ \texttt{OF} & \\ [] & : [] \\ [h] :: t & : E(h)'2 :: S_2(E,t) \end{array}$$

The function Π below built the list of indices of the first occurrences in the list l_2 of the elements in the list l_1.

$$\Pi(l_1,l_2) := \begin{array}{ll} \texttt{CASES}\ l_1\ \texttt{OF} & \\ [] & : [] \\ [h] :: t & : [search(h,l_2)] :: \Pi(t,l_2) \end{array}$$

Finally, the parallel reduction relation, $\rightarrow_{||E}$, is specified below. The functions S_1 and S_2 concatenate the left- and right-hand sides of the same equations in $E :: E^\circ :: \alpha$ given by the list of indices l.

$$s \rightarrow_{||E} t := \exists l : s = S_1(E :: E^\circ :: \alpha, l) \wedge t = S_2(E :: E^\circ :: \alpha, l)$$

The parallel congruence is defined as $\approx_{||E} := \rightarrow_{||E}^*$.

The congruence and the parallel congruence are the same relations. This is formalized as the Lemma 12.

Lemma 12 (Congruence versus parallel congruence).

$$(\approx_E) = (\approx_{||E})$$

Proving $(\approx_E) \subseteq (\approx_{||E})$: It is enough to prove $\rightarrow_E \subseteq \rightarrow_{||E}$.

Let s and t be strings such that $s \rightarrow_E t$.

This means that there exists s_1, t_1, w, u such that $s = ws'u$, $t = wt'u$ and $s' \approx t' \in E$.

This implies the existence of a list of indices l such that $s = S_1(E :: E^\circ :: \alpha, l)$ and $t = S_2(E :: E^\circ :: \alpha, l)$, where the list of indices l is built by selecting from α the indices of symbols to form the word w (adding twice the length of E); this list is concatenated with the index of the rule $s' \approx t'$ in E; and finally, the result is obtained concatenating the list of indices in α (adding twice the length of E) to form the word u.

To select from the list of equations α over the alphabet symbols in Σ the list of indices to form the words w and u, it is necessary to formalize some simple properties such as $\forall w \in \Sigma^* : S_1(\alpha, \Pi(w, \Sigma)) = S_2(\alpha, \Pi(w, \Sigma))$, where w and Σ are the list of symbols in w, and the list of alphabet symbols.

To conclude, since $\rightarrow_E \subseteq \rightarrow_{||E}$, closing both sides of the inclusion by reflexivity, symmetry, and transitivity, one obtains $(\approx_E) \subseteq (\approx_{||E})$.

Proving $(\approx_{||E}) \subseteq (\approx_E)$: It is necessary to prove $(\rightarrow_{||E}) \subseteq (\approx_E)$.

It implies to $\forall (s, t, l) : (s = S_1(E :: E^\circ :: \alpha, l) \wedge t = S_2(E :: E^\circ :: \alpha, l)) \implies s \approx_E t$.

The proof is made by induction for the length of the list l. Case l is an empty list, the lists s and t are also empty lists, and the assertive holds. Suppose, by the induction hypothesis, that the assertive holds for the list $l := l'$. Selecting a congruence from $E :: E^\circ :: \alpha$ through the index i, congruence can be selected from E, E° or α. Selecting from E, E°, or α, it results in: $(s_1 = w_1 S_1(E :: E^\circ :: \alpha, l') \wedge t_1 = w_2 S_2(E :: E^\circ :: \alpha, l')) \implies s_1 \approx_E t_1$ where $(E :: E^\circ :: \alpha)(i) = (w_1 \approx w_2)$. Applying the induction hypothesis, $w_1 a \approx_E w_2 b$ must hold, because the relation $\approx_E$ is closed by context. Thus, $(\rightarrow_{||E}) \subseteq (\approx_E)$ holds. Applying the reflexive-transitive closure in both sides of $\subseteq$, $(\approx_{||E}) \subseteq (\approx_E)$ holds.

The parallel reduction is used in the proof of the reduction from WP to PCP. The proof involves a simulation of the reduction through an instance of a solution of the PCP. But a solution in PCP can simulate parallel reduction, that implies that the reductions must be in parallel.

Let Σ be an alphabet. A domino or a piece has a form $\boxed{\frac{u}{w}}$ where $u, w \in \Sigma^*$.

A list of dominoes $D = \left[\boxed{\frac{u_1}{w_1}}, \cdots, \boxed{\frac{u_n}{w_n}}\right]$ and a list of indices $I = [i_1, \cdots, i_m]$ of D. The function A is defined as:

$$A(D, I) := \boxed{\frac{u_{i_1}}{w_{i_1}}} \bullet, \cdots, \bullet \boxed{\frac{u_{i_m}}{w_{i_m}}}$$

where $\boxed{\frac{u_j}{w_j}} \bullet \boxed{\frac{u_k}{w_k}} = \boxed{\frac{u_j u_k}{w_j w_k}}$.

A recursive definition of A is;

$$\begin{array}{rll} A(d,l) := & \texttt{CASES}\ l\ \texttt{OF} & \\ & [\,] & : \boxed{\frac{\square}{\square}}; \\ & [h] :: t & : d(h) \bullet A(d,t) \end{array}$$

The projections of a domino is defined as $\boxed{\frac{u}{w}}^{\triangleright} := u$ and $\boxed{\frac{u}{w}}_{\triangleright} := w$.

A domino list D has a solution if exists a non-empty list of valid indices I such that $A(D,I)^{\triangleright} = A(D,I)_{\triangleright}$.

Formally, $\mathit{Sol}(D) := \exists I : (\forall i < |I| : I(i) < |D|) \wedge A(D,I)^{\triangleright} = A(D,I)_{\triangleright}$.

Let Σ be an alphabet for a congruence of a Thue system. An extended alphabet Σ' is defined as:

$$\Sigma' := \{a, \overline{a} | a \in \Sigma\} \cup \{[,], \#, \overline{\#}\}$$

Let $s \approx^{?}_{E} t$ be a congruence over Σ. The transformations follow from this congruence, and is given through the following transformation rules.

$$\begin{array}{rcl} \overbrace{s} & := & \boxed{\frac{[s\#}{[}} \\ \underbrace{t} & := & \boxed{\frac{]}{\#\overline{t}]}} \\ \mathit{toDomino}_1(w \approx u) & := & \boxed{\frac{\overline{w}}{u}} \\ \mathit{toDomino}_2(w \approx u) & := & \boxed{\frac{w}{\overline{u}}} \\ \mathit{toDomino}_3(w \approx u) & := & \boxed{\frac{\overline{u}}{w}} \\ \mathit{toDomino}_4(w \approx u) & := & \boxed{\frac{u}{\overline{w}}} \\ \mathit{SThue2Dominoes}(E) & := & \mathit{map}(\mathit{toDomino}_1)(E) :: \mathit{map}(\mathit{toDomino}_2)(E) :: \\ & & \mathit{map}(\mathit{toDomino}_3)(E) :: \mathit{map}(\mathit{toDomino}_4)(E) \end{array}$$

$$
\begin{aligned}
alphadom_1 &:= \left[\boxed{\frac{c}{\overline{c}}} \mid c \in \Sigma\right] \\
alphadom_2 &:= \left[\boxed{\frac{\overline{c}}{c}} \mid c \in \Sigma\right] \\
(s \approx^?_E t)^{\boxed{\frac{d}{d}}} &:= \left[\overbrace{s}, \underbrace{t}, \boxed{\frac{\overline{\#}}{\#}}, \boxed{\frac{\#}{\overline{\#}}}\right] :: SThue2Dominoes(E) :: \\
&\qquad alphadom_1 :: alphadom_2
\end{aligned}
$$

Additionally, the lengths below are defined.

$length_1 := \left|\left[\overbrace{s}, \underbrace{t}, \boxed{\frac{\overline{\#}}{\#}}, \boxed{\frac{\#}{\overline{\#}}}\right] :: SThue2Dominoes(E)\right|$

$length_2 := \left|\left[\overbrace{s}, \underbrace{t}, \boxed{\frac{\overline{\#}}{\#}}, \boxed{\frac{\#}{\overline{\#}}}\right] :: SThue2Dominoes(E) :: alphadom_1\right|$

$L^{+k} := map(\lambda i \cdot i + k)(L)$

Then, the following lemmas are formalized.

Lemma 13 (). $\forall (D_1, D_2, I | \forall (i < |I|) : I(i) < |D_1|) : \; A(D_1 :: D_2, I) = A(D_1, I)$

Lemma 14 (). $\forall (D_1, D_2, I | \forall (i < |I|) : I(i) < |D_2|) : \; A(D_1 :: D_2, map(+|D_1|)(I)) = A(D_2, I)$

Both lemmas were formalized using induction in the length of I.

Lemma 15 (). $\forall (D, I_1, I_2) : \; A(D, I_1 :: I_2) = A(D, I_1) \bullet A(D, I_2)$

This lemma was formalized using induction in the length of I_1

The lemma below was formalized and it is part of the reduction of WP to PCP.

Lemma 16 (Congruence Implies in Dominoes Solution). Let $s \approx^?_E t$ be a congruence where s, t and for all $i < |E|$, $E(i)$ are formed by non-empty words. Thus, $s \approx^?_E t \implies Sol((s \approx^?_E t)^{\boxed{\frac{d}{d}}})$.

Starting an induction on the number of steps of $s \approx^?_E t$. But the induction can not be applied straightforwardly in the lemma. If a domino has a solution, there are infinite additional solutions that are not included in the book based on the formalization(Davis et al. [1994]). Depending on the kind of solution, it can disrupt the inductive step. The inductive step involves removing the first piece of the solution in the induction base and adding pieces to form a solution for the induction thesis. But if the first piece is also in

the middle of the solution list, it is necessary to replace each of these pieces in the middle with pieces of the solution of the induction thesis. To solve it, it is enough to formalize $s \approx^?_E t \implies (Sol((s \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overparen{s} \notin cdr((s \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overparen{s} = car((s \approx^?_E t)\boxed{\frac{d}{d}}))$. The assertive of the lemma is a corollary of it. This situation also occurs similarly in a proof of $Sol((s \approx^?_E t)\boxed{\frac{d}{d}}) \implies s \approx^?_E t$, but the solution is not simple as this lemma. To prove it is necessary to show that if a congruence transformed into a domino set has a solution, it has a minimal solution, i. e. a solution without repetition of the first piece.
Starting an induction, the basis is: $\approx^?_E = (\leftarrow_E \cup \rightarrow_E)^0$. In this case, $s = t$.
The solution is select by the list of indices $[0] :: \Pi(s, \Sigma)^{+length_2} :: [1]$, or using pieces:

$$\boxed{\frac{[s\#}{[}} \bullet \boxed{\frac{\overline{s}}{s}} \bullet \boxed{\frac{]}{\#\overline{s}]}} = \boxed{\frac{[s\#\overline{s}]}{[s\#\overline{s}]}}$$

Suppose that $s(\leftarrow_E \cup \rightarrow_E)^{k+1}t$. It is the same as stating that exists a word u such that $s(\leftarrow_E \cup \rightarrow_E)u$ and $u(\leftarrow_E \cup \rightarrow_E)^k t$. From $s(\leftarrow_E \cup \rightarrow_E)u$, it can be inferred that exists the words w_1, w_2, u_1 and u_2 such that $s = w_1u_1w_2$, $u = w_1u_2w_2$, and $(u_1 \approx u_2 \in E \vee u_2 \approx u_1 \in E)$. Without loss of generality, suppose that $u_1 \approx u_2 \in E$, that means that exists an index i such that $E(i) = (u_1 \approx u_2)$ By the induction hypothesis, $(Sol((u \approx^?_E u)\boxed{\frac{d}{d}}) \wedge \overparen{u} \notin cdr((u \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overparen{u} = car((u \approx^?_E t)\boxed{\frac{d}{d}}))$ Let I be the list of indices that select pieces to a solution in $(u \approx^?_E t)\boxed{\frac{d}{d}}$. The list of the indices of the pieces for the solution $(u \approx^?_E t)\boxed{\frac{d}{d}}$ could be selected by the list of indices $[0] :: \Pi(s, \Sigma)^{+length_1} :: [2] :: \Pi(w_1, \Sigma)^{+length_1} :: [2 \times |E| + i] :: \Pi(w_2, \Sigma)^{+length_1} :: [3] :: cdr(I)$. This list selects the pieces:

$$\boxed{\frac{[w_1u_1w_2\#}{[}} \bullet \boxed{\frac{\overline{w_1u_1w_2}}{w_1u_1w_2}} \bullet \boxed{\frac{\overline{\#}}{\#}} \bullet \boxed{\frac{w_1}{\overline{w_1}}} \bullet \boxed{\frac{u_1}{\overline{u_2}}} \bullet \boxed{\frac{w_2}{\overline{w_2}}} \bullet \boxed{\frac{\#}{\overline{\#}}} \bullet \boxed{\frac{\overline{w_1u_2w_2\#}\cdots t]}{\overline{\#}\cdots t]}}$$

$$= \boxed{\frac{[w_1u_1w_2\#\overline{w_1u_1w_2\#}w_1u_1w_2\#\overline{w_1u_2w_2\#}\cdots t]}{[w_1u_1w_2\#\overline{w_1u_1w_2\#}w_1u_1w_2\#\overline{w_1u_2w_2\#}\cdots t]}}$$

Above the piece $\boxed{\frac{\overline{w_1u_2w_2\#}\cdots b]}{\overline{\#}\cdots b]}}$ is selected by the list of indices $cdr(I)$.

The lemma below states that if a domino set has a solution it has a minimal solution.

Lemma 17 (Solution versus a Minimal Solution)**.** $Sol((s \approx^?_E t)\boxed{\frac{d}{d}}) \implies (Sol((s \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overbrace{s} \notin cdr((s \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overbrace{s} = car((s \approx^?_E t)\boxed{\frac{d}{d}}))$

Suppose $Sol((s \approx^?_E t)\boxed{\frac{d}{d}})$. Does it mean that exists a list of indices I of the domino set $(s \approx^?_E t)\boxed{\frac{d}{d}}$ such that $A((s \approx^?_E t)\boxed{\frac{d}{d}}, I)^{\triangleright} = A((s \approx^?_E t)\boxed{\frac{d}{d}}, I)_{\triangleright}$.
Let $s \sqsubset t$ be the word s is the suffix of the word t.
The following property holds for all I_1 and I_2:

$$(A((s \approx^?_E t)\boxed{\frac{d}{d}}, I_1 :: [0] :: I_2)^{\triangleright} \sqsubset A((s \approx^?_E t)\boxed{\frac{d}{d}}, I_1 :: [0] :: I_2)_{\triangleright}$$
$$\vee A((s \approx^?_E t)\boxed{\frac{d}{d}}, I_1 :: [0] :: I_2)_{\triangleright} \sqsubset A((s \approx^?_E t)\boxed{\frac{d}{d}}, I_1 :: [0] :: I_2)^{\triangleright})$$
$$\implies A((s \approx^?_E t)\boxed{\frac{d}{d}}, [0] :: I_2)^{\triangleright} = A((s \approx^?_E t)\boxed{\frac{d}{d}}, [0] :: I_2)_{\triangleright}$$

. Induction in the length of I_1 proves this property above.
The list of indices I can be split-ed into $I = I_1 :: [0] :: I_2$ such that $0 \notin I_2$
It implies that $[0] :: I_2$ selects a minimal solution in $(s \approx^?_E t)\boxed{\frac{d}{d}}$

The following properties are not formalized but here the complete proofs of the reduction from the undecidability of the Halting Problem to WP and WP to PCP were disscussed.

Lemma 18 (Dominoes Solution Implies in Congruence)**.** Let $s \approx^?_E t$ be a congruence where s, t and for all $i < |E|$, $E(i)$ are formed by non-empty words. Thus,

$$Sol((s \approx^?_E t)\boxed{\frac{d}{d}}) \implies s \approx^?_E t$$

The proof starts by induction on a number of pieces of the solution in $Sol((s \approx^?_E t)\boxed{\frac{d}{d}})$.

Actually, it does not work applying it straightforward in the assertion $Sol((s \approx^?_E t)\boxed{\frac{d}{d}}) \implies s \approx^?_E t$. But the induction works applied in the equivalent formula $(Sol((s \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overbrace{s} \notin cdr((s \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overbrace{s} = car((s \approx^?_E t)\boxed{\frac{d}{d}})) \implies s \approx^?_{||E} t$. Suppose $Sol((s \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overbrace{s} \notin cdr((s \approx^?_E t)\boxed{\frac{d}{d}}) \wedge \overbrace{s} = car((s \approx^?_E t)\boxed{\frac{d}{d}})$. A possible solution is:

$$\boxed{\frac{[s\#}{[}} \bullet \boxed{\frac{\bar{t}}{s}} \bullet \boxed{\frac{]}{\#\bar{t}]}} = \boxed{\frac{[s\#\bar{t}]}{[s\#\bar{t}]}}$$

.

This is the induction base and means that $s \rightarrow_{||E} t$.
If a solution does not have a form of the induction basis, it has a form:

$$\boxed{\frac{[s\#}{[}} \bullet \boxed{\frac{\bar{u}}{s}} \bullet \boxed{\frac{\overline{\#}}{\#}} \bullet \boxed{\frac{v}{\bar{u}}} \bullet \boxed{\frac{\#}{\overline{\#}}} \bullet \boxed{\frac{\cdots t]}{v\# \cdots t]}} = \boxed{\frac{[a\#\bar{u}\overline{\#}v\# \cdots t]}{[s\#\bar{u}\overline{\#}v\# \cdots t]}}$$

Taking out the first pieces, and add the piece $\boxed{\frac{[v\#}{[}}$, the solution below is a solution of $Sol((v \approx^?_E t)\boxed{\frac{d}{d}})$:

$$\boxed{\frac{[v\#}{[}} \bullet \boxed{\frac{\cdots t]}{v\# \cdots t]}} = \boxed{\frac{[v\# \cdots t]}{[v\# \cdots t]}}$$

By induction hypothesis, $v \approx^?_{||E} t$ holds. The assertions $s \rightarrow_{||E} u$ and $u \rightarrow_{||E} v$ also hold because the instances $\boxed{\frac{\bar{u}}{s}}$ and $\boxed{\frac{v}{\bar{u}}}$ can be built. This property that if the instances $\boxed{\frac{s}{\bar{t}}}$ and $\boxed{\frac{\bar{s}}{t}}$ can be built implies that $s \rightarrow_{||E} t$ is proved by induction in the number of pieces uses to build them and only works in a parallel reduction relation. Thus, $s \approx^?_{||E} t$.

Corollary 6 (WP reduces to PCP)**.** Let $s \approx^?_E t$ be a congruence where s, t and for all $i < |E|$, $E(i)$ are formed by non-empty words. Thus, $s \approx^?_E t \Leftrightarrow Sol((s \approx^?_E t)\boxed{\frac{d}{d}})$.

This is a corollary of the lemmas 16 and 18.

In the proof in the textbook (Davis et al. [1994]) the authors use a computable function to reduce from Turing machines to a question of congruence over a Thue system. It works well in proof assistance Coq (Forster et al. [2018]) (Heiter [2017]) because by the Calculus of the Inductive Constructions, that Coq is based on, only allows total computable functions Larchey-Wendling [2017].

In PVS, which allows non-computable functions, it is possible to build a non-computable reduction function that proves the undecidability of the WP. For example, the next lemma states that there exists a function to reduce the Halting problem to WP.

Lemma 19 (Halting Problem reduces to WP ↗)**.** There exists a reduction function ω from `partial_recursive` to a triple $\langle E_T \times \Sigma^* \times \Sigma^* \rangle$, where E_T is the type of the reduction rules, such that: $\forall(pvs_0 : \texttt{partial\_recursive}) : T_\varepsilon(pvs_0) \Leftrightarrow \omega(pvs_0)'2 \approx_{\omega(pvs_0)'1} \omega(pvs_0)'3$

Building the reduction function as below and analyzing each case in the if-then-else statement is enough to prove it: $\omega(pvs_0) := \left\langle [], a^{\kappa_p(pvs_0)}, \begin{array}{l} \texttt{IF } T_\varepsilon(pvs_0) \texttt{ THEN } a^{\kappa_p(pvs_0)} \\ \texttt{ELSE IF } \kappa_p(pvs_0) > 0 \texttt{ THEN } \square \\ \texttt{ELSE } a \end{array} \right\rangle$
Above $a^{\kappa_p(pvs_0)}$ is the word of the symbol a repeated $\kappa_p(pvs_0)$ times.

The reduction function ω is not a partial recursive function. But the proof of the undecidability of the WP requires that the reduction function belongs to the class of the terminating partial recursive functions. There is no easy way to solve this situation. PVS supports the execution of computable functions through a feature called PVSio. Maybe, PVS must be improved to support separating types of computable and non-computable functions. For example, the function `f` specified below must be of the non-computable type because it is deciding if the value `n` belongs to the intersection of the sets `s1` and `s2` or if it belongs to `s1` or `s2` exclusively, and according to the corollary 5, deciding if an element belongs to a set of naturals is undecidable.

```
f(n: nat, s1 : set[nat], s2 : set[nat]
    | union(s1,s2) = fullset[nat]) : below[3] =
        IF s1(n) AND s2(n) THEN 2
        ELSEIF s1(n) THEN 1
        ELSE 0 ENDIF
```

The separation of the computable and non-computable functions is useful for the specification of the following theorem.

Theorem 7 (Undecidability of the WP)**.** For all PVS computable bijective function β from $\langle \Sigma^* \times \Sigma^* \times E_T \rangle$ to a natural number, there no exists a `computable` program *oracle* such that:

$$(\neg\gamma(\mathit{oracle})(\beta(s,t,E),\bot)) \Leftrightarrow s \approx^{?}_{E} t$$

.

The formalization of the correspondence between WP and PCP may be applied in cryptography. Both problems are used to create one-way functions when applied to represent Turing Machines with their inputs (Kozhevnikov and Nikolenko [2009]); for instance, WP can be used to implement an authentication protocol in which agents show that they know a solution to a problem without showing it. Such an authentication protocol is described through the steps below.

1. The agent A generates a list E of rewrite rules over a binary alphabet, words u_1 and u_2 such that $u_1 \approx_{||E} u_2$.
2. The agent B knows that only the agent A knows the solution of $u_1 \approx_{||E} u_2$. Then, by the insecure channel, B asks A to prove if she/he knows the solution (without showing the solution).
3. Finally, A generates a list of twenty words congruent modulo E to u_1 and sends them to B. Then, B asks for a solution for each word in the list, according to the answer given by A, B verifies each of them and decides to answer if either it is a solution for u_1 or for u_2.

Only those who know the solution of $u_1 \approx_{||E} u_2$ can answer it correctly and, supposing that for each word in a list, the chance to know it is $\frac{1}{2}$, the chance to know all the solutions for the elements of the list is less than one in one million because $(\frac{1}{2})^{20} = \frac{1}{1048576}$.

A non-possibility of an algorithm to solve WP avoids the secret solution of the protocol being found by an attacker.

Chapter 7

Related Work

Nowadays, mechanical proofs concerning computability properties serve as a formalization exercise and hold significant importance in offering formal feedback to practical computational models. Therefore, the primary objective of the `PVS0` models, both single and multiple-function, as stated in the introduction, is to create automation mechanisms for verifying the termination of PVS programs through the application of different criteria Alves Almeida [2021], Alves Almeida and Ayala-Rincón [2020], Muñoz et al. [2021].

The publication (Ferreira Ramos et al. [2018]) highlights that `SF-PVS0` programs have restrictions on inductive levels. For example, level zero enables specifications using only basic functions such as successor, greater-than, and projections. In contrast, subsequent levels allow the specification of other `computable` functions that can be called operators. Developing the composition of such `SF-PVS0` programs was challenging, which hindered the formalization of concepts like Turing completeness and Rice's Theorem. However, Chapter 2 demonstrates that composing programs specified in the `MF-PVS0` language is uncomplicated, as it involves using the offset operator $\_^{+}\_$.

When it comes to the `SF-PVS0` language, creating a third program for composing two programs (not necessarily terminating) is not feasible in a general sense due to its reliance on the combinatorial structure of the input programs. The issue was resolved swiftly during the proof of the undecidability of the Halting Problem, as only particular composition constructions (assuming termination of functions) were necessary (Ferreira Ramos et al. [2018]). However, the current work overcomes such difficulties by employing a language that supports programs consisting of several functions capable of recursive calls, both to themselves and each other.

In addition, in (Ferreira Ramos et al. [2018]), the `SF-PVS0` model was analyzed, and the formalization of the equivalencies between semantical termination criteria was achieved. Although the equivalencies between termination criteria have been formalized for the `SF-PVS0` model as discussed in (Ferreira Ramos et al. [2018]) and (Muñoz et al. [2021]), the MF-PVS0 model lacks such formalization. All the termination criteria mentioned in the introduction (Turing or TCC termination (Turing [1949]); size-change principle (Lee et al. [2001]); calling context graphs (Manolios and Vroon [2006]); matrix weighted graphs (Avelar [2015]); dependency pairs (Arts and Giesl [2000]), (Alves Almeida [2021]), (Alves Almeida and Ayala-Rincón [2019])) apply to both models (Muñoz et al. [2021]). However, some termination criteria mentioned above require modifications to accommodate static analysis of multiple-function programs that allow mutual recursion (which is avoided in the PVS functional specification language).

One way to verify termination properties in MF-PVS0 programs is by converting them into SF-PVS0 programs. A possible area for future improvement is enhancing PVS and developing an automated translation from PVS recursive functions to SF-PVS0 programs. This task would incorporate the functions invoked by a recursive PVS function specification into the built-in operators' lists within the SF-PVS0 model. This approach is consistent with the PVS specification discipline, which necessitates verifying the well-definedness of all elements within a PVS function before verifying the function itself.

Formalizing computability properties has been a common practice since the inception of theorem provers and proof assistants. Examples of such formalizations include the mechanical proof of the undecidability of the Halting Problem in (Boyer and Moore [1984]), where the model of computation is the LISP language. Another instance is the formalization of the same undecidability outcome in Agda, as reported in (Johannisson [2000]), where the model of computation is based on axioms defined over the elements of an abstract type called Prog.

Recent work in formalizing computability result over computational models related to lambda calculus, and programming languages are the focus here. For example, Forster and Smolka (Forster and Smolka [2017]) used call-by-value lambda calculus as the model of computation, which is Turing complete and has beta-reduction applicable only to a beta-redex that is not below an abstraction with an argument being an abstraction. Forster and Smolka formalized various computational properties, including Rice's Theorem, which states that semantic predicates containing both elements and their complements are non-recognizable. Norrish (Norrish [2011]) also formalized Rice's Theorem and other properties, such as the existence of universal machines and a version of the *s-m-n* Theorem, using HOL4 for the lambda calculus model. In Lean, Carneiro (Carneiro [2019]) formalized Rice's Theorem using partial recursive functions as the model of computation, with the Fixed-Point Theorem used to derive it as a corollary. The work also yielded the undecidability of the Uniform Halting Problem as a corollary.

The current work differs from Carneiro's (Carneiro [2019]), Norrish's (Norrish [2011]), and Forster and Smolka's (Forster and Smolka [2017]) works in terms of the model of computation. For example, lambda calculus enables a straightforward formalization of Rice's Theorem due to the simplicity of implementing necessary operators to simplify the proof, such as a fixed-point operator. In contrast, the Recursion Theorem plays the fixed point construction role in the formalization of Rice's Theorem for the `MF-PVS0` model, which requires the Gödelization technique. Furthermore, using partial recursive functions or a concrete functional language model like `MF-PVS0` programs enables the formalization of functional program properties, such as those related to their complexity and termination criteria. The latter objective motivated the specification of the `PVS0` language.

As previously mentioned, the choice of the computation model affects the selection of problems used for reductions and the complexity of formalizations. For instance, when formalizing the undecidability of the Word Problem over monoids, using Turing Machines as in Post's classical approach (Post [1947]) seems natural. This approach constructs a reduction from the Halting Problem (over Turing Machines) to the Word Problem. Several authors have reported similar formalization approaches for other word problems, such as PCP (e.g., (Forster et al. [2018]), (Heiter [2017])).

Post's approach represents configurations of Turing Machines as words and the transition function as reductions by a Thue system. The atomicity of Turing Machine transitions allows each transition to be represented as a rewriting rule. However, proving the Word Problem over the `MF-PVS0` model is much more complex since reducing the Halting Problem of `MF-PVS0` programs to the Word Problem is not straightforward. The non-atomicity of the evaluation steps of `MF-PVS0` programs makes constructing such a reduction challenging.

For example, different evaluations of $\kappa_2^S(\texttt{cnst}(4), \texttt{cnst}(7))$ using the built-in operators from section 4.1 would require a representation of the constants and the symbol κ_2^S as a word (over some alphabet). Additionally, it requires a Thue system that simulates the evaluation of κ_2, which involves evaluating other operators such as addition, multiplication, and division by two. The complexity of proofs of undecidability properties generally depends on the chosen computational model.

Ongoing efforts are being made to formalize the undecidability of the Post Correspondence Problem (PCP), which is not directly linked to the properties of a specific computational model. In (Hopcroft et al. [2006]) (Chapter 12), it is explained how a function can be employed to reduce the Word Problem (WP) for Thue systems to PCP. This approach is followed in the formalization discussed in Chapter 6. The reduction function converts a congruence question into a question about a "domino set" by duplicating the alphabet, adding two separator letters, and generating domino pieces for each rewriting rule, the word in the congruence question, and the letter in the new alphabet. The PVS specification captures this function, ensuring that a congruence question on the

Word Problem implies its formalization as a PCP question through this transformation. However, it should be stressed that the fact that an instance of PCP gives a WP congruence question (lemma 18) and the undecidability of WP were not formalized. To formalize the undecidability of WP, it is possible to specify a PVS function that reduces from the Halting Problem of `MF-PVS0` programs to WP, i.e., that transforms a `MF-PVS0` program into a WP instance such that this WP instance simulates the execution of the `MF-PVS0` program. Then, supposing that WP is decidable, a contradiction arises because it implies the decidability of the Halting Problem.

The unique choice of our formalization is the reduction from the Recursion to Rice's Theorem. Although there are well-known properties used in textbook proofs, such as the existence of a universal machine or the undecidability of the universal language (e.g., (Sipser [2012]), (Hopcroft et al. [2006])), complete formalizations of computational properties do not follow from such constructions to the best of our knowledge. Instead, other strategies are followed, such as reductions from the Fixed-Point Theorem in (Carneiro [2019]), Rice's Lemma in (Forster and Smolka [2017]), and the Recursion Theorem in the current work.

To prove the undecidability of the Halting Problem, Fixed-Point, Recursion, and Rice's Theorem, the effort required to formalize them directly over the `MF-PVS0` computational model is similar to that needed for other models. If we were to translate these proofs to another computational model, we would need to formalize the properties and then put in additional effort to prove the conservativeness of the translation. When it comes to properties not directly related to the computational model, such as the undecidability of PCP and the Word Problem discussed earlier, selecting an appropriate computational model is essential. In addition, it may be worthwhile to specify other models and develop verified conservative translations between them for such cases.

The Recursion Theorem, using the built-in operators like κ_2, successor, and projections composed to κ_2^{-1} and greater-than, provides the most straightforward approach to formalize Rice's Theorem for the `MF-PVS0` model. It is important to note that for a Turing complete model like `MF-PVS0`, the Fixed-Point and recursion Theorems are interchangeable, as one can be proven from the other.

Other interesting computability results have been formalized using linguistic computational models. For example, Forster, Kirsk, and Smolka (Forster et al. [2019]) utilized Coq's constructive type theory to synthesize a method for formalizing the undecidability of first-order formulas' validity, satisfiability, and provability. Forster and Larchey-Wendling employed Coq to formalize a reduction from the Post Correspondence Problem (PCP) to the provability of intuitionistic linear logic, utilizing binary stack machines and Minsky machines in the process (Forster and Larchey-Wendling [2019]). The authors accomplished this through a chain of reductions that began with the PCP, moved on to binary PCP, binary PCP with indices, binary stack machines, Minsky machines, and finally, intuitionistic linear logic provability. Finally, in Larchey-Wendling [2017], Larchey-Wendling utilized Coq to formalize that the type $\mathbb{N}^k \to \mathbb{N}$ encompasses all k-ary recursive functions that can be proven total in Coq, including the set of primitive recursive functions.

Representing the class of partial recursive functions in Coq using the type $\mathbb{N}^k \to$ `option` $\mathbb{N}$ would not be feasible. In PVS, the operator `Maybe`, which adds $\diamondsuit$ to the working type, is used instead of the functor `option`. Attempting to use the type $\mathbb{N}^k \to$ `option` $\mathbb{N}$ to represent partial recursive functions in Coq fails because the decidability of totality contradicts the undecidability of the Halting Problem. Therefore, a similar approach to the one used by the semantic evaluation function χ of `MF-PVS0` programs can be taken in Coq to deal with this issue. That is, partial recursive functions can be represented by a predicate of type $\mathbb{N}^k \to \mathbb{N} \to$ `Prop`.

Functional types in PVS enable the specification of non-recursive functions, such as the PVS function $non\_comp(n)$ with the type $\mathbb{N} \to \mathbb{N}$, defined below:

$non\_comp(n) := max(v_o : \mathbb{N}; |; \gamma(\kappa_e^{-1}(\kappa_2^{-1}(n)'1))(\kappa_2^{-1}(n)'2, v_o) \cup -1) + 1.$

This function takes a natural number n that represents a `MF-PVS0` program and an input (respectively, $\kappa_e^{-1}(\kappa_2^{-1}(n)'1))$ and $\kappa_2^{-1}(n)'2)$, and is well-defined in PVS. This happens because the maximum of finite sets is well-defined, and by the determinism of γ, the set $v_o : \mathbb{N}; |; \gamma(\kappa_e^{-1}(\kappa_2^{-1}(n)'1))(\kappa_2^{-1}(n)'2, v_o)$ is either empty or singleton. To ensure the model is not trivially Turing Complete, the built-in operators of the `MF-PVS0` model must be fixed adequately. This restriction allows the specification of non-computable functions while avoiding a model over a set of non-recursive operators.

Recently, Larchey-Wendling and Forster presented a formalization of the undecidability of Hilbert's Tenth Problem (Larchey-Wendling and Forster [2022]). The authors accomplished this by utilizing a series of problem reductions, including the Halting Problem for Turing Machines, the Post Correspondence Problem, a specialized Halting Problem for Minsky Machines, the termination of FRACTAN (a language model dealing with register machines), and finally, solvability of Diophantine logic and Diophantine equations.

Finally, Spies and Forster (Spies and Forster [2020]) and Kirst and Larchey-Wendling (Kirst and Larchey-Wendling [2022]) made significant contributions to the Coq library of synthetic undecidability proofs by mechanizing two interesting results. The first result pertains to the undecidability of higher-order unification, while the second deals with the undecidability of first-order satisfiability by finite models (FSAT). Goldfarb's proof of the undecidability of second-order unification is formalized in Coq, and from this proof, the undecidability of higher-order unification follows as a corollary. This work is discussed in (Goldfarb [1981]).

Chapter 8

Future Work and Conclusion

This thesis formally investigates the functional-language computational models using two models, `SF-PVS0` and `MF-PVS0` with the support of the proof assistant PVS. Formal proofs of computational properties are formalized in PVS. A formalization of Turing Completeness of `MF-PVS0` is established for a subset of partial recursive `MF-PVS0` programs, built using fundamental operators for successor, projection, greater-than functions, and bijective operators to encode tuples from naturals and vice versa. The mechanization in PVS involves verifying the correctness of `MF-PVS0` implementations of these functions and operators for composition, primitive recurrence, and minimization.

In addition, Rice's Theorem for the `MF-PVS0` model is formalized. The proof of Rice's Theorem uses the Recursion Theorem and a Gödelization of `MF-PVS0` programs. Furthermore, it relies on Cantor's diagonal argument to derive a contradiction from a `MF-PVS0` program that can decide semantic predicates about other `MF-PVS0` programs. To prove Rice's Theorem using the Recursion Theorem, selecting any Gödelization is possible whenever it is the same for both the Recursion Theorem and Rice's Theorem. Several corollaries of Rice's Theorem are also formalized, including the undecidability of the uniform Halting Problem, the functional equivalence problem, the existence of fixed points problem, self-replication, and the subset problem for natural numbers. Finally, the development includes formalizations of the undecidability of the Halting Problem and the Fixed-Point Theorem for `MF-PVS0`, as well as part of the reduction from the Word Problem to the Post Correspondence Problem.

Formalizing the undecidability of the Halting Problem for both the `SF-PVS0` and `MF-PVS0` models necessitates the ability to compute the bijective function κ_2 from tuples of naturals to naturals (defined in the equation 3.1). However, the specific Gödelization function utilized may vary between different functions mapping `MF-PVS0` programs to natural numbers.

This portion of the `PVS0` project contributed 369 proven lemmas, of which 230 are

Type Correctness Conditions (TCCs), proof obligations generated automatically by PVS but not necessarily proved automatically. Table 8.1 provides quantitative information on the proof files. The data from auxiliary theories that did not require significant work is not included in the totals given in the table.

Table 8.1: Relevant quantitative data

PVS theory	Lines of Code (loc) of proof files 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	35
`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
`pcp_string_rewriting`	102175 loc - 1,8M	10	34
`pcp`	1799 loc - 76K	4	8
`string_rewriting`	6223 loc - 367K	26	6
`pvs0_halting`	605 loc - 32K	2	3

The formalization of Turing Completeness may result in more extensive proofs than the Recursion Theorem, but its formalization is simpler overall. The auxiliarly lemmas (and proofs) required for Turing Completeness deal with technical and straightforward issues that require semantic evaluation, such as expansions of the predicate ε and simple instantiations of existentially quantified premises. Additionally, some auxiliary theories necessitate a significant number of lemmas, including `mf_pvs0_expr` and `mf_pvs0_lang`, which contain proofs related to the operational semantics of the `MF-PVS0` language, and `mf_pvs0_computable`, which includes results concerning Gödelizations.

Several other theorems would be valuable to formalize for the `PVS0` language models, including the *s*-*m*-*n* theorem, Post's Theorem, the existence of a universal machine, the linear speedup theorem, the tape compression theorem, the time hierarchy theorem, the space hierarchy theorem. However, the primary challenge and intriguing aspect of such formal developments in the `PVS0` models is that the traditional proofs of these theorems rely on specificities of machine models, such as Turing machines and register machines, and abstract language models, such as the Lambda calculus. Moreover, it would be even more fascinating to formalize other undecidability results that go beyond the context of properties of computational models, such as the Word Problem for algebraic structures (Post [1947]), PCP, and Hilbert's Tenth Problem (Larchey-Wendling and Forster [2022]).

Two recent developments worth mentioning are the formalizations of Post's theorem for weak call-by-value lambda calculus and the undecidability of the PCP through the reduction of the Halting Problem for Turing machines, both achieved using Coq Forster and Smolka [2017], Forster et al. [2018]. While the functional model `PVS0`, lambda calculus, and Turing machines have certain similarities that make them helpful in formalizing such theorems for `PVS0` programs, significant challenges are still involved. The formalizations are closely tied to their respective computational models, and creating the necessary translations is not straightforward.

The undecidability of WP and PCP was addressed because connecting the computability properties of the `PVS0` models to properties not directly related to computability is relevant. Some of the necessary lemmas to complete the WP and PCP formalizations were formalized. For example, the equivalence between the congruence and parallel congruence relations in WP, the existence of minimal solutions for solvable domino sets, and transformations of congruence solutions into domino set solutions. However, the formalization of the undecidability of WP and PCP is an ongoing work that requires, in addition to fulfilling the proof of the lemma of the reduction from WP to PCP, establishing a reduction of `PVS0` programs related to WP instances. Such a reduction should take a `MF-PVS0` program and transform the question of its termination into a Thue system whose rules simulate the execution of the `MF-PVS0` program.

References

Ariane Alves Almeida. *On Termination by Dependency Pairs and Termination of First-Order Functional Specifications in PVS*. PhD thesis, Universidade de Brasília, Graduate Program in Informatics, Brasília, Distrito Federal, Brazil, 2021. URL `https://repositorio.unb.br/handle/10482/42296`. 2, 75, 76

Ariane Alves Almeida and Mauricio Ayala-Rincón. Formalizing the dependency pair criterion for innermost termination. *CoRR*, abs/1911.00406, 2019. URL `http://arxiv.org/abs/1911.00406`. 76

Ariane Alves Almeida and Mauricio Ayala-Rincón. Formalizing the dependency pair criterion for innermost termination. *Sci. Comput. Program.*, 195:102474, 2020. URL `https://doi.org/10.1016/j.scico.2020.102474`. 2, 75

Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000. URL `http://dx.doi.org/10.1016/S0304-3975(99)00207-8`. 76

Andréia B. Avelar. *Formalização da automação da terminação através de grafos com matrizes de medida*. PhD thesis, Universidade de Brasília, Departamento de Matemática, Brasília, Distrito Federal, Brazil, 2015. URL `https://repositorio.unb.br/handle/10482/18069`. In Portuguese. 2, 3, 76

Frédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.*, 21(4):827–859, 2011. URL `https://doi.org/10.1017/S0960129511000120`. 1

Robert Stephen Boyer and J Strother Moore. A Mechanical Proof of the Unsolvability of the Halting Problem. *Journal of the Association for Computing Machinery*, 31(3): 441–458, 1984. URL `https://doi.org/10.1145/828.1882`. 76

Mario 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. URL `https://doi.org/10.4230/LIPIcs.ITP.2019.12`. 76, 78

Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. The ACL2 Sedan Theorem Proving System. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th*

International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, volume 6605 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2011. URL `https://doi.org/10.1007/978-3-642-19835-9_27`. 2

M. Davis, R. Sigal, and E.J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Computer Science and Scientific Computing. Elsevier Science, 1994. ISBN 9780122063824. URL `https://books.google.com.br/books?id=6G_arEqHtysC`. 64, 69, 72

Martin D. Davis. *Computability and Unsolvability*. McGraw-Hill Series in Information Processing and Computers. McGraw-Hill, 1958. 63

Thiago Mendonça Ferreira Ramos, César Augusto Muñoz, Mauricio Ayala-Rincón, Mariano Miguel Moscato, Aaron Dutle, and Anthony Narkawicz. Formalization of the Undecidability of the Halting Problem for a Functional Language. In *25th International Workshop on Logic, Language, Information, and Computation WoLLIC*, volume 10944 of *Lecture Notes in Computer Science*, pages 196–209. Springer, 2018. URL `https://doi.org/10.1007/978-3-662-57669-4_11`. 3, 16, 19, 55, 75

Robert W. Floyd and Richard Beigel. *The Language of Machines: An Introduction to Computability and Formal Languages*. W H Freeman & Co, 1994. URL `https://doi.org/10.2307/2275690`. 4, 59

Yannick Forster and Dominique 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. URL `https://doi.org/10.1145/3293880.3294096`. 78

Yannick Forster and Gert 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 *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017. URL `https://doi.org/10.1007/978-3-319-66107-0_13`. 76, 78, 82

Yannick Forster, Edith Heiter, and Gert Smolka. Verification of PCP-Related Computational Reductions in Coq. In *9th International Conference on Interactive Theorem Proving ITP*, volume 10895 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2018. URL `https://doi.org/10.1007/978-3-319-94821-8_15`. 72, 77, 82

Yannick Forster, Dominik Kirst, and Gert 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. URL `https://doi.org/10.1145/3293880.3294091`. 78

Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981. URL `https://doi.org/10.1016/0304-3975(81)90040-2`. 79

Edith Heiter. Undecidability of the Post Correspondence Problem. Master's thesis, Faculty of Mathematics and Computer Science, Saarland University, 2017. URL `https://www.ps.uni-saarland.de/~heiter/downloads/PCP_Undecidability.pdf`. 72, 77

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 3rd edition, 2006. ISBN 0321455363. 64, 77, 78

Vasile I Istratescu. *Fixed point theory: an introduction*. Springer, 1981. 57

Kristofer Johannisson. Formalizing the Halting Problem in a Constructive Type Theory. In *International Workshop on Types for Proofs and Programs TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 2000. URL `https://doi.org/10.1007/3-540-45842-5_10`. 76

Dominik Kirst and Dominique Larchey-Wendling. Trakhtenbrot's Theorem in Coq: Finite Model Theory through the Constructive Lens. *Log. Methods Comput. Sci.*, 18(2), 2022. URL `https://doi.org/10.46298/lmcs-18(2:17)2022`. 79

A. A. Kozhevnikov and Sergey I. Nikolenko. On complete one-way functions. *Probl. Inf. Transm.*, 45(2):168–183, 2009. URL `https://doi.org/10.1134/S0032946009020082`. 74

Dominique Larchey-Wendling. Typing Total Recursive Functions in Coq. In *8th International Conference on Interactive Theorem Proving ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2017. URL `https://doi.org/10.1007/978-3-319-66107-0_24`. 73, 79

Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in coq (extended version). *Log. Methods Comput. Sci.*, 18(1), 2022. URL `https://doi.org/10.46298/lmcs-18(1:35)2022`. 64, 79, 82

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2001. 2, 76

Salvador Lucas. The origins of the halting problem. *J. Log. Algebraic Methods Program.*, 121:100687, 2021. URL `https://doi.org/10.1016/j.jlamp.2021.100687`. 63

Panagiotis Manolios and Daron Vroon. Termination Analysis with Calling Context Graphs. In *Computer Aided Verification, 18th International Conference, CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2006. URL `https://doi.org/10.1007/11817963_36`. 2, 3, 76

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-Level Control Through Deep Reinforcement Learning. *Nature*, 518(7540): 529–533, 2015. 57

César A. Muñoz, Mauricio Ayala-Rincón, Mariano M. Moscato, Aaron Dutle, Anthony J. Narkawicz, Ariane Alves Almeida, Andréia B. Avelar, and Thiago Mendonça Ferreira Ramos. Formal Verification of Termination Criteria for First-Order Recursive Functions. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 27:1–27:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL `https://doi.org/10.4230/LIPIcs.ITP.2021.27`. 2, 3, 75, 76

César A. Muñoz, Mauricio Ayala-Rincón, Mariano M. Moscato, Aaron Dutle, Anthony J. Narkawicz, Ariane Alves Almeida, Andréia B. Avelar, and Thiago Mendonça Ferreira Ramos. Formal Verification of Termination Criteria for First-Order Recursive Functions - Journal Version. *J. Autom. Reason.*, 2023. In press. Accepted in the JAR Special Issue of ITP 2021, May 2023. 2, 3

Michael Norrish. Mechanised Computability Theory. In *Second International Conference on Interactive Theorem Proving ITP*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011. URL `https://doi.org/10.1007/978-3-642-22863-6_22`. 76

Sam Owre and Natarajan Shankar. The formal semantics of PVS. TR SRI CSL-97-2, SRI International, Menlo Park, March 1999. URL `http://www.csl.sri.com/papers/csl-97-2/`. 6

Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992. URL `https://doi.org/10.1007/3-540-55602-8_217`. 6

Étienne Payet, David J. Pearce, and Fausto Spoto. On the termination of borrow checking in featherweight rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 411–430. Springer, 2022. URL `https://doi.org/10.1007/978-3-031-06773-0_22`. 1

Emil L. Post. Recursive unsolvability of a problem of Thue. *The Journal of Symbolic Logic*, 12(1):1–11, 1947. URL `https://doi.org/10.2307/2267170`. 63, 77, 82

Thiago Mendonça Ferreira Ramos, Ariane Alves Almeida, and Mauricio Ayala-Rincón. Formalization of the Computational Theory of a Turing Complete Functional Language Model. *J. Autom. Reason.*, 66(4):1031–1063, 2022. URL `https://doi.org/10.1007/s10817-021-09615-x`. 19

J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: predicate subtyping in pvs. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998. doi: 10.1109/32.713327. 7

Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, third edition, 2012. URL `https://doi.org/10.1145/230514.571645`. 4, 38, 64, 78

Simon Spies and Yannick Forster. Undecidability of higher-order unification formalised in coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 143–157. ACM, 2020. URL `https://doi.org/10.1145/3372885.3373832`. 64, 79

René Thiemann and Christian Sternagel. Certification of Termination Proofs Using CeTA. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics TPHOL*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. doi: 10.1007/978-3-642-03359-9_31. 2

Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937a. URL `https://doi.org/10.1112/plms/s2-42.1.230`. 63

Alan Mathison Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937b. URL `https://doi.org/10.2307/2268280`. 4

Alan Mathison Turing. Checking a large routine. In *Report of a Conference High Speed Automatic Calculating-Machines*, pages 67–69. University Mathematical Laboratory, 1949. 76