Formalizing Abstract Computability: Turing Categories in Coq

Polina Vinogradova[a,1] Amy P. Felty[a,b,2] Philip Scott[b,a,3]

[a] *School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada*

[b] *Department of Mathematics and Statistics, University of Ottawa, Ottawa, Canada*

Abstract

The concept of computable functions (as developed by Gödel, Church, Turing, and Kleene in the 1930's) has been extensively studied, leading to the modern subject of recursive function theory. However recent work by category theorists has led to a more conceptual and abstract foundation of computability theory—Turing categories. A Turing category models the notion of partial map as well as recursive computation, using methods of categorical algebra to formalize these concepts. The goal of this work is to provide a formal framework for analyzing this categorical model of computation. We use the Coq Proof Assistant, which implements the Calculus of (co)Inductive Constructions (CIC), and we build on an existing Coq library for general category theory. We focus on both formalizing Turing categories and on building a general framework in the form of a well-structured Coq library that can be further extended. We begin by formalizing definitions, propositions, and proofs pertaining to Turing categories, and then instantiate the more general Turing category formalism with a CIC description of the category which explicitly models the language of partial recursive functions.

Keywords: Category theory, Turing categories, Computability, Formalization, Calculus of Inductive Constructions (CIC), Coq proof assistant

1 Introduction

Traditional computation theory (Gödel, Turing, Kleene) originally aimed at capturing the informal notion of computable functions over the natural numbers and computable theories of real numbers (e.g., [25,15]). Even before Turing, Church's introduction of lambda calculus [4] attempted to capture certain properties of computation in a broader sense, through manipulation of strings of symbols representing function application and abstraction. It was subsequently shown that the numerical (partial) functions computable by these various abstract formalisms for computation coincided, thus suggesting the so-called Church-Turing thesis. Later independent formalisms for defining numerical computation (e.g., Register Machines, Markov's

[1] Email: polina.vino@gmail.com. Research partially supported by NSERC Discovery Grants of Felty and Scott.

[2] Email: afelty@uottawa.ca. Research partially supported by an NSERC Discovery Grant.

[3] Email: phil@site.uottawa.ca. Research partially supported by an NSERC Discovery Grant.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

algorithms, etc.) were shown to again lead to the same class of computable numerical partial functions, lending yet more credence to the Church-Turing thesis. Nevertheless, practical as well as theoretical computer science requires more than just numerical computation: one has increasingly abstract theories of computation over various data types, higher-order computation, computation based on various programming language paradigms, newer paradigms of computation (parallel, probabilistic, quantum), etc. Category theory appears to be both general and expressive enough to be the tool of choice for modeling computation in these newer senses.

Many concepts of traditional partiality, recursion and computation theory have begun to be effectively analyzed categorically. Some early work on expressing partiality in terms of category theory includes papers of Robinson and Rosolini [23] as well as Mulry [20]. More recent studies on partiality, relevant to our work on Turing categories, are mentioned below. Categorical analysis of computation and recursion ranges from early fundamental work of Elgot on flowchart semantics, and its connections with denotational semantics (cf. Manes and Arbib [19]), to analysis of Church's theories of lambda calculi (both typed and untyped) in cartesian closed categories, associated higher-order categorical logics and domain theory [16,1]. In a series of works of increasing categorical generality, beginning with Longo and Moggi [18], Di Paola and Heller [22] and culminating in recent work of Cockett and Hofstra [9], we see the beginnings of a new and direct categorical development of the foundations of recursion theory

This paper is based on the thesis of the first author [28]. The model we study is the category-theoretic formalism of *Turing Categories*, introduced by Cockett and Hofstra [9]. Turing categories are a very general computational model, built from a categorical analysis of partial maps (Restriction Categories) by Cockett and Lack [10]. The partial maps of a Turing category arise as the computable maps of a partial combinatory algebra (PCA) [17]. Moreover, recent work establishes criteria for determining when various complexity classes of total maps can be made into a Turing category [5]. Thus the notion of Turing category provides a robust, abstract framework for discussing computation over a wide range of settings.

Our study of the Turing category computation model takes the form of building a type-theoretic formal language description (formalization) of the relevant concepts. The concepts we have selected to formalize lay the groundwork for (formally) proving abstract interpretations of standard theorems in recursion theory. The key motivation behind this approach is the level of organization, consistency, and guaranteed correctness it provides in working with proofs and definitions for which unformalized presentations may omit important and interesting details.

Turing category theory can be viewed as an (up until recently) non-formalized mathematical framework that describes a precise model of computation. As computation on a physical computer is a precise procedure, it seems natural to verify that a formal description of this framework exactly fits the selected categorical model. This is the motivating idea and the main objective of this work. There is not a huge amount of work done in this direction of research; specifically, in formalizing a category as an instance of an abstract computational model. Furthermore, we choose to work in the Coq Proof Assistant, with the Calculus of (co)Inductive Constructions (CIC) as its underlying formal language. Thus, we are using intuitionistic logic to

build the proofs and definitions in this formalization. This further differentiates this development from traditional recursion theory, and adds interesting constructivist information to our proofs. For example, to verify if $f : A \to B$ is a function, we must confirm that for each proof that $x \in A$, we can prove $f(x) \in B$.

There have been previous attempts to formalize certain aspects of computation, both as categorically abstracted concepts and as direct formalizations of partial or total computation. Our project, in fact, builds on an existing constructive formalization of partial recursive functions in Coq [30], and makes use of the S_n^m theorem proved within the resulting language. There are other formalizations of traditional computation, such as primitive recursion in [21], a weak call-by-value lambda calculus as a model of computation in Coq [13], a formalization of computable functions done directly using lambda abstractions (rather than a specific proof assistant, although the project was motivated by considerations of NuPRL) [11], and formalizations of computability theory done in different proof assistants such as HOL [29]. (The latter formalization is done using non-constructive logic). But we stress that our formalization (in Coq) is based on the novel structure of Turing categories, and an associated theory of partial maps in categories.

As far as formalization of categorical abstractions of computational structures goes, a formalization (using Coq) of cartesian closed categories, which have been previously used to model total computation, is found in [24]. Furthermore, a formalization of a categorical partiality structure which represents the same notion that we use as the partiality structure in Turing categories has also previously been done using the Agda proof assistant [3].

We start from a library for general category theory developed by Timany and Jacobs [24], designed to take advantage of advanced features in Coq 8.5 such as type classes and universe polymorphism. This library successfully develops many of the basic concepts, and thus we chose to adopt the style of definitions and formalization strategy used in this library. With this library as a starting point, we specify the mathematical definitions found in the framework of the Turing Category computation model, as well as abstract versions of other types of structures naturally occurring in the traditional computation model. We then formally prove (the abstract versions of) a number of results from traditional recursion theory.

In addition to formalizing the categorical concepts, we formalize several examples of categories. These examples provide validation of our formalization approach and formalized results. They also provide a mechanism to formally study these specific example categories. Our main example is the formalization of traditional computation on the natural numbers and the categorical interpretation of all the structure found therein, illustrating that these indeed conform to the Turing category model formalism. We base our formalization of traditional recursion theory on a formalization due to Zammit [30].

Our Coq scripts, compilation instructions, and a link to the library we build on is available at: https://github.com/polinavino/Turing-Category-Formalization.

2 Our Formalization

We divide our explanation of the formalization according to the Coq files we have built. The section corresponding to each file discusses definitions and their formal encoding as well as the challenges of reconciling the differences between them. We show only small examples of Coq code. The reader is referred to the online repository for the full code. For an expository introduction to the Restriction and Turing Categories below, the reader is referred to R. Cockett's notes and slides: [6,7] as well as the original papers [10,9].

2.1 Cartesian Restriction Categories and their Formalization

In order to model partial recursive functions in the framework of Turing categories, we use a particular characterization of categories of partial maps. These were introduced in Cockett and Lack [10], under the name *restriction categories*.

Restriction categories are based on the idea of a *restriction combinator*. The latter associates to a map $f : A \to B$ an idempotent (its restriction) $\overline{f} : A \to A$ whose axioms categorically capture key aspects of the "domain" of f (sufficient for characterizing categories of partial maps.)

R1 $f\overline{f} = f$

R2 $\overline{f}\overline{g} = \overline{g}\overline{f}$ whenever $\mathrm{dom}(f) = \mathrm{dom}(g)$

R3 $\overline{g\overline{f}} = \overline{g}\overline{f}$ whenever $\mathrm{dom}(f) = \mathrm{dom}(g)$

R4 $\overline{g}f = f\overline{gf}$ whenever $\mathrm{cod}(f) = \mathrm{dom}(g)$

We can also capture *totality* in this context, by defining a morphism $f : A \to B$ to be a *total map* if $\overline{f} = 1_A$.

Examples of restriction categories include familiar categories of partial functions (formalized in Sections 2.5 and 2.6 below) as well as the category of topological spaces and partial continuous functions with open domains. Continuous functions which preserve open sets, called *open maps*, have also been axiomatized abstractly and play an influential role in computer science [14]. This allows topological notions to be lifted to other categories and forms an important part of the theory of restriction categories [8] (see also Section 2.4 below).

In addition to such partial map structure, a category in which we wish to axiomatize computation requires a version of cartesian products which interact in a meaningful way with the restriction structure. These partial versions of products and terminal objects are called restriction (or partial) products and restriction-terminal objects, respectively. A category which admits these structures is called a *cartesian restriction category* ([10]), and may be considered as a partial map version of *cartesian categories*, i.e. categories with finite cartesian products and a terminal object, as in [16].

Following the style of the Coq category theory library we have selected, we use type classes to formalize categorical notions. Type classes are a versatile and convenient way to encapsulate terms and propositions about them into a single term representing its informal counterpart, with a number of features particularly useful for reasoning about category theory.

```
Class Category: Type :=
{
  (* Type of Objects *)
  Obj: Type;

  (* Type of morphism beween two objects *)
  Hom: Obj -> Obj -> Type;

  (* composition of morphisms: *)
  compose: forall {a b c: Obj}, Hom a b -> Hom b c -> Hom a c
           where "f o g" := (compose g f);

  (* associativity of composition: *)
  assoc: forall {a b c d: Obj} (f: Hom a b) (g: Hom b c) (h: Hom c d),
         ((h o g) o f) = (h o (g o f));

  (* symmetric form of associativity: *)
  assoc_sym: forall {a b c d: Obj} (f: Hom a b) (g: Hom b c) (h: Hom c d),
             ((h o (g o f) = (h o g) o f));

  (* identity morphisms: *)
  id: forall {a: Obj}, Hom a a;

  (* id left unit: *)
  id_unit_left: forall (a b: Obj) (h: Hom a b), id o h = h;

  (* id right unit: *)
  id_unit_right: forall (a b: Obj) (h: Hom a b), h o id = h
}.
```

Fig. 1. The Coq definition of the `Category` type class

To illustrate the general approach, Figure 1 contains the definition of `Category`, which comes from the library, and is defined as a type class, while Figure 2 contains some of our code that uses it. The first two type declarations in Figure 1 define Coq identifiers for objects and morphisms of a category, along with their types, while the third defines the composition operator with its (dependent) type, along with some notation for it. Thus terms of type `Obj` correspond to the objects of a category, and given `a` and `b` of type `Obj`, terms of type `Hom a b` are arrows between the corresponding objects. The curly brackets in the definition of `compose` indicate that the first three arguments can be omitted because they can be inferred from the types of the last two arguments, i.e, (`compose a b c g f`) can be written (`f o g`). The next two declarations state two forms of associativity of composition. These become proof obligations when defining a new category and instantiating it with particular values for `Obj`, `Hom`, and `compose`. Similarly an identity morphism is declared with its type, followed by two proof obligations about it.

In Figure 2, the first declaration provides the type of the restriction combinator; it takes a map in a given category, and returns another map from the source object of the original map to itself. It is used in the definition of the restriction combinator, which declares `rc` to be an operator of this type. In addition, in the body of this class, there are also terms `rc1`, ..., `rc4`, the types of which correspond to the axioms that `rc` must satisfy. These axioms are formalized versions of the axioms R1, ..., R4 discussed above. The lemma that follows (proof omitted) states a property that follows from these axioms. Next, `RestrictionCat` is defined as a type class whose structure combines its two arguments—a category and a restriction combinator defined in that category—where `rc_d` represents the proof of the lemma for these arguments.

A similar approach is used to define the notions of a total subcategory, trivial restriction structure, etc. [9]. For cartesian restriction structure, we formalize restriction products and restriction-terminal objects, which are similar in structure to

```
Definition rcType (C: Category): Type := forall a b: C, @Hom C a b -> @Hom C a a.

Class RestrictionComb (C: Category): Type :=
{
  rc: rcType C;
  rc1: forall (a b: Obj) (f: Hom a b), f o (rc a b f) = f;
  rc2: forall (a b c: Obj) (f: Hom a b) (g: Hom a c),
         (rc a c g) o (rc a b f) = (rc a b f) o (rc a c g);
  rc3: forall (a b c: Obj) (f: Hom a b) (g: Hom a c),
         rc a c (g o (rc a b f)) = (rc a c g) o (rc a b f);
  rc4: forall (a b c: Obj) (f: Hom a b) (g: Hom b c),
         (rc b c g) o f = f o (rc a c (g o f))
}.

Lemma rc_d_pf: forall (C: Category) (RC: RestrictionComb C) (a b c: Obj)
                      (f: Hom a b) (g: Hom b c),
  rc a c (g o f) = rc a b ((rc b c g) o f).

Class RestrictionCat (C: Category) (rc: RestrictionComb C): Type :=
{
  RCat_RC: RestrictionComb C := rc;
  rc_d := rc_d_pf C RCat_RC
}.
```

Fig. 2. Some Coq code for restriction categories

true products and true terminal objects. We have formalized restriction products and the restriction terminal object following closely the example of how the true products and the true terminal object are defined in the category theory library we are using. In order to define a cartesian restriction category, we build a class that takes as parameters a category, a restriction combinator in this category, partial products for all pairs of objects in the category, and the partial terminal object.

We note a key fact, related to the constructive viewpoint taken here. Subcategories in the library we have selected are made up of objects and arrows which are distinct from those in the larger category. Indeed, the objects and arrows in a subcategory are *pairs* consisting of the object (or arrow) in the larger category *together with a proof* that the given object (or arrow) is indeed a member of the subcategory. So in order to define a subcategory, predicates that are true for objects (or arrows) that are to be included in the subcategory must first be defined. Following that, a proof that this selection of objects and arrows indeed forms a subcategory is required to complete the subcategory definition.

We have formalized and proved a number of standard results about the cartesian restriction structure we built, including the following (all of which come from the original papers [10,9]):

(i) The total maps in a restriction category form a subcategory, called the *total subcategory* (more precisely, the *subcategory of total maps*);

(ii) A restriction terminal object in a cartesian restriction category is a (true) terminal object in its total subcategory;

(iii) Restriction products in a cartesian restriction category are (true) products in its total subcategory;

(iv) In an embedding-retraction pair (m, r), m is a total map.

We will not go into details discussing the finer points in the above list, but we will elaborate on (i) to give the reader the flavour of the process of formalizing these kinds of category theory proofs. We will define a total subcategory term `Tot` which takes an existing restriction category as a parameter and outputs a total subcategory structure on the given category. This means we must obtain a subcategory which

contains all the objects of the larger category and only those arrows for which the restriction is equal to the identity, i.e., those maps $f : A \to B$ such that $\overline{f} = 1_A$.

To define such a subcategory `Tot` using the terms in the category theory library (over which we are building our formalization), we must first define a predicate `TotMaps` having the following type:

```
forall (rc: RestrictionComb) (RC: RestrictionCat)
       (a b: Obj), Hom a b -> Prop.
```

It is defined so that `(TotMaps rc a b f)` holds when `rc f` = `id a`, where `id a` is the identity map on `a`. Next, we define the desired (total) subcategory to be the term obtained by applying `Wide_SubCategory` (a library-defined term for constructing *wide subcategories*: these are subcategories which contain all objects of the larger category and a subcollection of the maps) to two arguments: the original restriction category and the predicate `TotMaps rc RC`.

To complete the instantiation of the total subcategory, we must prove that for every object `a : Obj`, the identity `id a` is indeed contained in the resulting wide subcategory, and also that it contains the composition of any two total maps. The idea of the formal proofs of these claims is to use the proof part of the dependent types representing maps in this category, as well as Coq rewriting tactics to fulfill the proof obligations.

2.2 Turing Categories and their Formalization

A Turing category is a cartesian restriction category that contains a special kind of structure that models computation. We say a category T is *Turing* if it contains an object $A \in \mathsf{T}$, called a *Turing object*, and a family of "application" morphisms for A, $\{\tau_{X,Y} : A \times X \to Y \mid X, Y \in \mathsf{T}\}$ with the weak universal property that every morphism $Z \times X \to Y$ factors through $\tau_{X,Y}$ via some total "curried" map $Z \to A$. This is similar to the factorization of maps in a cartesian closed category (CCC) (see [16]), if we think of A (by analogy) as a kind of (partial) function space Y^X. One can prove that a Turing object A in a Turing category is a *universal object*, in the sense that every object in the category is a retract of it.

An alternative, equivalent presentation of Turing categories is to consider a cartesian restriction category with a universal object A, as above, together with a single distinguished self-application map $\bullet : A \times A \to A$, called a *Turing morphism* satisfying a similar weak universal property to above.

We have formalized Turing structure along with a number of standard results (taken from [9]), including:

(i) Every object in a Turing category is a retract of a Turing object (i.e. the Turing object is a *universal object*);

(ii) A CCC with trivial restriction structure and an object A of which every object is a retract is a Turing category;

(iii) An object B in a Turing category with Turing object A is Turing if and only if it is a retract of A;

(iv) The halting domain is m-complete;

(v) The equivalent characterization of Turing categories in terms of a Turing morphism and object embeddings.

2.3 Partial Combinatory Algebras and Related Categories

The underlying computational mechanism of a Turing category is given by a fundamental structure arising in combinatory logic and untyped lambda calculus: a combinatory algebra [1,17]. Following Cockett and Hofstra [9], we work in the general setting of cartesian restriction categories. Let C be such a category. A *partial combinatory algebra* (PCA) in C is a pair $\mathbb{A} = (A, \bullet)$ consisting of an object $A \in \mathsf{C}$ and a map $\bullet : A \times A \to A$ satisfying a partial version of combinatory (or functional) completeness [1,17].

We have formalized the definition of a PCA, which also required formalizing n-fold products of A, as well as isomorphisms between equal powers of A. We assume that whenever $n_1 + m_1 = n_2 + m_2$, $A^{n_1} \times A^{m_1} \cong A^{n_2} \times A^{m_2}$. [4]

Consider a PCA $\mathbb{A}$. Following [9], we consider the category $\mathrm{Comp}(\mathbb{A})$, the smallest cartesian restriction subcategory of C on the objects $1, A, A^2, \cdots$ containing every total map $1 \to A$, as well as the application $\bullet$. Any morphism $f : A^n \to A^m$ in this subcategory is called a *polynomial map*. If $\mathbb{A}$ is combinatory complete, such polynomial maps coincide with a class of maps called $\mathbb{A}$-*computable maps* $A^n \to A^m$. An $\mathbb{A}$-computable map f is essentially given by a total constant (code) $c_f : 1 \to A$ acting by iterated $\bullet$. A key result of [9] is that $\mathrm{Comp}(\mathbb{A})$ is a Turing Category, with Turing morphism $\bullet$.

We have formalized $\mathrm{Comp}(\mathbb{A})$ and $\mathrm{Split}(\mathrm{Comp}(\mathbb{A}))$ (the Karoubi envelope of $\mathrm{Comp}(\mathbb{A})$, i.e. the idempotent-splitting completion of the category $\mathrm{Comp}(\mathbb{A})$ [16]) as cartesian restriction categories. The resulting terms can be instantiated to give specific instances of $\mathrm{Comp}(\mathbb{A})$ and $\mathrm{Split}(\mathrm{Comp}(\mathbb{A}))$ when supplied with arguments including the base category $\mathtt{C}$, the object $\mathtt{A} : \mathtt{C}$, as well as cartesian restriction structure in $\mathtt{C}$ (which, by our construction, will be inherited by both of the resulting categories). In the process of building these terms, we have discovered additional conditions required to show intuitionistically that $\mathrm{Split}(\mathrm{Comp}(\mathbb{A}))$ is a cartesian restriction category. In addition, we have formalized the relationship between a Turing category T with a Turing object A and the related categories $\mathrm{Comp}(\mathbb{A})$ and $\mathrm{Split}(\mathrm{Comp}(\mathbb{A}))$, as well as the proofs of the following result: there are inclusions (fully faithful embeddings) $\mathrm{Comp}(\mathbb{A}) \hookrightarrow \mathsf{T} \hookrightarrow \mathrm{Split}(\mathrm{Comp}(\mathbb{A}))$. [5]

2.4 Range Categories and their Formalization

A key notion in traditional computability theory is the study of recursively enumerable (r.e.) sets of numbers. Such sets arise as the range of a total recursive function. We will be interested in adding range structure to restriction categories. Ranges can be expressed in terms of another type of combinator which (whenever

[4] We have not fully formalized the proof of this isomorphism. Note that the $A^0 = 1$ case cannot be treated as a term of the same type as A^n with $n > 0$. A formalization of these properties of cartesian products is not critical for the project discussed here. From our progress so far, it is clear that it is straightforward to do so, although it requires a tedious and detailed recursive proof, where the boundary cases are especially hard to negotiate.

[5] The reader is referred to [28], p.28 for a summary, and Section 5.5.3, pp. 100-103 for its formalization.

it exists) is in a sense dual to the restriction combinator [8]. A *range combinator* is an operator $\widehat{(-)}$ that takes a map $f : A \to B$ to a map $\widehat{f} : B \to B$, satisfying the following axioms:

RR.1 $\quad \widehat{\overline{\widehat{f}}} = \hat{f}$

RR.2 $\quad \widehat{f}f = f$

RR.3 $\quad \widehat{\overline{g}f} = \overline{g}\widehat{f}$ for all maps f, g with $cod(f) = dom(g)$

RR.4 $\quad \widehat{g\widehat{f}} = \widehat{gf}$ for all maps f, g with $cod(f) = dom(g)$

The maps $\widehat{f}$ are obviously idempotent. Open maps as presented in [14] allow an abstract characterization of ranges. For example, it is shown in [8] that a restriction category C is a range category if and only if every map is open.

We have chosen to formalize this particular abstraction because in the process of formalizing the motivating examples, it became apparent that representing partiality using a total formal language presented one of the biggest challenges as well as one of the greatest curiosities. We have formalized a number of results regarding the interactions between range structure and embedding-retraction pairs, as well as a criterion for a Turing category to admit cartesian range structure, coming from [27].

For instance, we have formalized the key conditions under which range structure interacts well with Turing structure:

(i) For every idempotent, there exists an equivalent idempotent (i.e., has the same splitting) which is equal to its own range;

(ii) Applying the range combinator over a (partial) product of two maps is a distributive operation (the Beck-Chevalley condition).

Note that without existing range structure in a category, one cannot express the above conditions in terms of a range combinator. However, in light of the above-mentioned connection of range categories with open maps, it turns out we can express these conditions in terms of open maps instead. In particular, the key formal results (among other things) must be expressed strictly in terms of openness of point maps $1 \to A$ and the $\bullet$ map, in a given Turing category T. This fact was overlooked in the original informal proof, and the proof had to be restructured via open maps in order to be formalized. This is an example of the types of challenges and insights that we have encountered during the process of formalizing categorical definitions and proofs.

Continuing the use of open maps, the final result of Section 2.3 can be extended to range categories as follows. Suppose T is a Turing category with Turing object A. Suppose the universal application $\bullet : A \times A \to A$ as well as every point $1 \to A$ are open and the Beck-Chevalley condition holds. Then Comp($\mathbb{A}$) , T, and Split(Comp($\mathbb{A}$)) are range categories, and Comp($\mathbb{A}$) $\hookrightarrow \mathsf{T} \hookrightarrow$ Split(Comp($\mathbb{A}$)) are range-preserving inclusions. [6]

Next, we discuss formalizing the motivating examples of these categorical structures.

[6] The above has been formalized and proved in [28], Section 5.6.2. This transits through a direct translation of range structure in terms of combinators, contained in [27].

```
Instance Par_Cat : Category :=
{
  Obj := Set;
  Hom := fun (A B : Set) => {P:A -> Prop & (forall x:A, P x -> B )};
  compose := fun (A B C : Set) (F : Hom A B) (G: Hom B C) =>
             existT ...
  id := fun (A:Set) => existT ... (fun _ => True) (fun (x:A) (_:True) => x).
}.
```

Fig. 3. Outline of Coq code defining `Par_Cat`

2.5 Formalizing the Category of Sets and Partial Maps

The category Par (of Sets and partial maps) is the motivating example for the categorical structure discussed above, including cartesian restriction and cartesian range structure, but not including Turing or PCA structure. We define the formal version of Par, which we call `Par_Cat`, as an instance of the `Category` type class from the original Coq library (i.e., we must define the required objects, morphisms, proofs of associativity, etc.). Figure 3 contains an outline of the structure of this category in Coq. As usual, we must fill in the type of objects, which in this case is simply `Set`, followed by the type of morphisms. Due to the total nature of computation in CIC, it is impossible to directly represent a partial map. For this reason, we must define the type of the set of all partial maps from A to B quite differently from the type of all total maps from A to B in the category Set. This latter category is formalized by the Set category type class instance defined in the original library, `Set_Cat`. As an aside, we also formally define and prove that there is an equivalence of categories between Set and the total subcategory Tot(Par). However, a *partial map* from `A` to `B` is a dependent pair consisting of a domain predicate $\texttt{P} : \texttt{A} \rightarrow \texttt{Prop}$ together with a map of type $\texttt{forall x} : \texttt{A}, \texttt{P x} \rightarrow \texttt{B}$, which takes two arguments: an "element" `x` of the set `A`, and a proof of the proposition `P x`.

In the definition of the identity function `id`, the dependent pair constructor `sigT` is used to pair the always true proposition (the identity is total) with the function mapping both an element `x` of `A` and a proof that `x` satisfies this trivial proposition to `x`. An underscore (_) is used to elide the names of parameters when they are not used in the body of the definition. We write ... where we omit some typing information. The type of the `compose` operator is also shown in the figure, but the definition is omitted except to show the use of the dependent pair constructor. Its definition involves correctly composing both the function applications and the two proofs that the domain predicates are satisfied. Immediately following the code in the figure the required properties about `compose` and `id` must be proved.

After the definition of this category and its required proofs, we can instantiate the restriction combinator. The restriction of a given map $\texttt{f} : \texttt{Hom a b}$ has the same domain predicate `P` as the map `f` itself, but has the restriction of `f` evaluated at $\texttt{x} : \texttt{A}$ and a proof $\texttt{pf} : \texttt{P x}$. Note that once we have defined the restriction combinator mapping, we must also prove that it satisfies the required axioms in order to complete the instantiation.

More specifically, we are not using the language to represent a specific (explicitly defined) map, but rather we must be able to compare two arbitrary maps, presenting a proof of an equality judgment. To implement such equality proofs for partial maps in Par and its subcategories, one must formalize such notions as Kleene equality (and compare it with Leibnitz equality), as well as various aspects of proof irrelevance

and functional extensionality (see [28], pp. 116-119). To simplify such proofs, an axiomatic presentation was adopted in [28].

Following a similar format, we have also instantiated Par as a cartesian restriction category and a cartesian range category (i.e., defined all the required maps, objects, and completed the accompanying proofs).

2.6 Formalizing the Category of Partial Recursive Maps

The category Par contains a subcategory of maps that are partial recursive, i.e., computable by a map which can be expressed in terms of the partial recursive constructors (zero, successor, projection, recursion, substitution and minimalization [12]). [7] We use this definition of formal computation as the basis for our formalization of the category of computable maps. The motivating example is Comp($\mathbb{N}$). We build our subcategory using an existing formalization of this presentation of computation as well as the proof of the S_n^m theorem completed using this definition [30].

This formalization gives the definition and the semantics of the language of partial recursive maps separately. We define the language constructors as an inductive type called `prf`, defined as follows in Coq:

```
Inductive prf : Type :=
| Zero : prf
| Succ : prf
| Proj : nat -> prf
| Sub : prf -> prf -> nat -> nat -> prf
| Rec : prf -> prf -> prf
| Min : prf -> prf.
```

The semantics are given as an inductively-defined relation, whose header is:

```
Inductive converges_to : prf -> list nat -> nat -> Prop
```

where (`converges_to f_prf ln n`) is provable whenever, informally, the partial recursive function `f_prf` applied to the list of natural numbers `ln` outputs `n`. Note that this "output" is unique, so we are able to build a partial map in the `Par_Cat` sense, described above, which corresponds to a given `prf` term.

In order to build a map that is Kleene-equal to the computation that a particular `f_prf` : `prf` represents, we use the `converges_to` relation to give the domain predicate, and add a special case of an axiom of choice to select, given a list `ln` : `list nat`, a natural number as output *in the case when there is a proof that this output necessarily exists* (i.e., a proof `pf` : `exists n, converges_to f_prf ln n`).

Now, we formally consider a partial map with m components as a map in the category of partial recursive maps whenever there is a proof that each of its components is Kleene-equal to a `prf` computation. This category inherits cartesian restriction structure defined in the larger category, `Par_Cat`. In order to demonstrate Turing structure in this category, we take the natural numbers to be the Turing object, and (given a Gödel enumeration $\{\phi_n\}$ of all computable functions) the application

[7] Although the results in this subsection are formally proved, they depend on a collection of appropriate axioms. These should be straightforward to prove in a more detailed analysis, which may involve clarification of the axioms of equality used in this work.

map $\bullet : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by $\bullet(n, x) = \phi_n(x)$ to be the Turing map. We express this map formally in terms of the constructors in the `prf` language. This is a formal analog of the so-called Kleene PCA $(\mathbb{N}, \bullet)$.

Next, we must prove two more results to show that we have built a Turing structure in the category of computable maps. First, we show the resulting map `bullet : prf` does in fact constitute Turing structure, i.e., prove that for all `n : nat` and `x : nat`, this map satisfies the $\bullet(n, x) = \phi_n(x)$ equation expressed in terms of the `converges_to` predicate.

Second, it is required that the `bullet` map is itself computable by a map that exists in the `prf` language (which corresponds to the *universal application* in traditional recursion theory). Formally completing this proof is a key part of formalizing the category of computable maps because a map in this subcategory $\mathbb{N}^n \to \mathbb{N}^m$ is, like a map in any subcategory in our formalization, a pair of a map in the larger category and a proof that it is indeed contained in the subcategory. In this case, after defining `bullet`, we must show its computability explicitly.

We have formally proved that the necessary categorical diagrams commute (as maps in the larger `Par_Cat` category), thus demonstrating that we have indeed built a Turing category.

3 Discussion and Future Work

In this project, we have accomplished our two primary goals:

(i) building a Coq library which formalizes Turing categories as well as some key concepts useful for studying abstract computation using this formal categorical framework;

(ii) formalizing the motivating examples for each of the categorical concepts we have formalized as instances of the corresponding type classes within the framework we have built.

The categorical concepts on which the tools we developed in (i) are based provide us with the advantage of allowing us to model both partial and total computation constructively. The tool framework itself is integrated into an existing comprehensive category theory Coq library, and it conforms to the structure existing therein.

Such a formal development enhances the traditional study of abstract computation by introducing structural and hierarchical integrity and making precise all definitions. Furthermore, building our tools as an extension of an existing library facilitates future research on the topic by using and building on the categorical structure and results of the library.

As with most proofs in category theory, proving results about Turing, restriction and range categorical structure do not require reasoning using the law of excluded middle, or any other application specifically of classical logic. Thus, proofs of the category-theoretic results we chose to formalize hold in a constructive setting. We note that we did use several specific versions of the Axiom of Choice in our formalization to help structure our definitions and make them compatible with the Coq library that we have used. They are used in limited sections of the library. While in general, in impredicative intuitionist type theory (higher-order logic), the Axiom

of Choice implies the Law of Excluded Middle ([16], pp. 160-164), the versions considered here are much weaker. They are closer in spirit to either Russell's theory of definite descriptions or the existence property.

While most of the formal results we have proved confirm what has already been shown in the literature, formalization also gives us the ability to find omissions in the definitions, proofs and propositions. For example, in the process of formalizing a result about ranges in Turing categories, we saw that we were not able to directly express the result in terms of range structure, and had to instead formulate and prove a very closely related result in terms of open maps (see Section 2.4).

In (ii), we formalize partial maps and partial recursion using (strongly normalizing and intuitionistic) CIC, and study how the resulting formalisms can be forced to conform to the categorical framework we have built. It is in this part of the project that we really see the advantages of formally representing partial maps and recursion abstractly, such as not having to model partial maps using total functions or relations (or total functions built using relations).

In the formalization of the category of sets and partial maps (as well as its subcategories), it is not always the case that we can build definitions (and therefore proofs) directly following the strategies in the documents we are working from. For example, because of the way formalized partial maps are structured, in order to prove equality between partial maps formally, we require a stronger version of the (dependent) functional extensionality axiom as well as the proof irrelevance axiom (to identify all proofs of the same proposition as equal). In general, the formalization of concepts related to partiality, such as the range of a partial map, expressed both categorically and extensionally (in terms of sets and maps), was one of the most interesting aspects of our formal study. For this reason, we chose to focus our formalization work more heavily on the study of partiality.

The most noteworthy result we have formalized is the constructive version of the category of sets of the form $\mathbb{N}^n$ and partial recursive maps between them, which is meant to categorically represent traditional computation. This has not previously been done. Through our work, we have gained an understanding (as well as formal constructions) of the additional results, concepts and machinery that are needed to build such a category. In the process of building this category in the formal sense, we have also encountered several proofs whose completion necessitated the use of specific versions of the Axiom of Choice, as mentioned earlier.

Here, again, there are certain repercussions of not being able to use the law of excluded middle, such as constructing partial maps out of a language of partial recursive maps `prf` (discussed in the previous section), the semantics of which can most directly be expressed as a relation. Some of the key recursion theory results have already been demonstrated using this language directly (such as the S_n^m theorem) [30], and therefore hold in the (cartesian restriction) Turing category we have built out of this language. However, the purpose of Turing categories is, in part, to be able to do as much recursion outside of the extensional reasoning of set theory as possible. That is, we wish to get away from studying recursion in terms of points, i.e. maps of the form $1 \to A$, for an object A in some category. It would be interesting to extend the scope of our formal framework to include other categorical structures, for example equalizers, (co)monads, and higher-order structure, and

formalize their role in abstract categorical recursion theory (e.g. [16,17]).

There are a number of other promising directions for further applying this framework. The most natural, perhaps, is the formalization of the Turing category-formulated abstraction of Rice's theorem. This will require the formalization of a number of general categorical concepts such joins and meets, as emphasized in Cockett's lectures [7]. Such general concepts are widely applicable to other results as well.

Applying our framework in another direction, it would be an interesting and innovative pursuit to use it to formalize computational complexity classes of total maps in Turing categories, as presented in [5]. Other potentially interesting options for building on this framework include formalizing monoidal Turing categories (with differential structure) and conducting a formal study more focused on the PCA's (which, recall, are computation-modeling structures at the core of every Turing category) as well as relationships between them.

As an alternative to formalizing computation in the mainstream Coq development, one could also explore formalization of Turing categories and extensional examples thereof using the version of Coq that implements homotopy type theory [2]. This development of the Coq system eliminates the need for explicitly adding equality axioms due to reformulating the concept of equality in the underlying theory, thus promising much more natural ways to identify equal functions in a category. There do exist other proof assistant systems implementing the mathematical ideas underlying homotopy type theory, such as Lean [26], which does so in a more integrated way than the Coq development. The Lean system has two modes, the proof irrelevant mode (which is incompatible with homotopy type theory), and the homotopy type theory mode. This system is another option to consider for future work on formalizing categorical examples of abstract computation.

References

[1] Amadio, R. M. and P.-L. Curien, "Domains and Lambda-Calculi," Cambridge Tracts in Theoretical Computer Science **46**, Cambridge University Press, 1998.

[2] Bauer, A., J. Gross, P. L. Lumsdaine, M. Shulman, M. Sozeau and B. Spitters, *The HoTT library: A formalization of homotopy type theory in coq*, CoRR **abs/1610.04591** (2016).
URL http://arxiv.org/abs/1610.04591

[3] Chapman, J., T. Uustalu and N. Veltri, *Formalizing restriction categories*, Journal of Formalized Reasoning **10** (2017).

[4] Church, A., "The Calculi of Lambda Conversion." Annals of Mathematics Studies, Princeton University Press, 1941.

[5] Cockett, J., P. Hofstra and P. Hrubeš, *Total maps of Turing categories*, Electronic Notes in Theoretical Computer Science (Proceedings MFPS XXX) **308** (2014), pp. 129–146.

[6] Cockett, R., *Categories and computability* (2010), full Course Notes and Slides,15th Estonian Winter School in Computer Science (EWSCS).
URL https://pages.cpsc.ucalgary.ca/~robin

[7] Cockett, R., *Turing categories and computability* (2010), slides from 15th Estonian Winter School in Computer Science (EWSCS).
URL http://cs.ioc.ee/ewscs/2010/cockett/estonia-slides-4.pdf

[8] Cockett, R., X. Guo and P. Hofstra, *Range categories II: Towards regularity*, Theory and Applications of Categories **26** (2012), pp. 453–500.

[9] Cockett, R. and P. Hofstra, *Introduction to Turing categories*, Annals of Pure and Applied Logic **156(2-3)** (2008), pp. 183–209.

[10] Cockett, R. and S. Lack, *Restriction categories I*, Theoretical Computer Science **270** (2002), pp. 223–259.

[11] Constable, R. L. and S. F. Smith, *Computational foundations of basic recursive function theory*, Theoretical Computer Science **121** (1993), pp. 89–112.
URL http://www.sciencedirect.com/science/article/pii/0304397593900858

[12] Cutland, N., "Computability: An introduction to recursive function theory," Cambridge University Press, 1980.

[13] Forster, Y. and G. Smolka, *Weak call-by-value lambda calculus as a model of computation in coq*, in: *Proceedings of the 8th International Conference on Interactive Theorem Proving* (2017), pp. 189–206.
URL https://doi.org/10.1007/978-3-319-66107-0_13

[14] Joyal, A., M. Nielsen and G. Winskel, *Bisimulation from open maps*, Information and Compututation **127** (1996), pp. 164–185.
URL http://dx.doi.org/10.1006/inco.1996.0057

[15] Kleene, S., "Introduction to Metamathematics," Ishi Press International, 2009.

[16] Lambek, J. and P. J. Scott, "Introduction to Higher Order Categorical Logic," Cambridge Studies in Advanced Mathematics, Cambridge University Press, 1986.

[17] Longley, J. and D. Normann, "Higher-Order Computability," Springer, 2015.

[18] Longo, G. and E. Moggi, "Gödel numberings, principal morphisms, combinatory algebras," Springer Berlin Heidelberg, 1984 pp. 397–406.

[19] Manes, E. and M. Arbib, "Algebraic Approaches to Program Semantics," Monographs in Computer Science, Springer New York, 1986.

[20] Mulry, P. S., *Partial map classifiers and partial cartesian closed categories*, Theoretical Computer Science **136** (1994), pp. 109–123.
URL http://www.sciencedirect.com/science/article/pii/0304397594001242

[21] O'Connor, R., "Incompleteness and Completeness: Formalizing Logic and Analysis in Type Theory," Ph.D. thesis, Radboud University Nijmegen (2009).

[22] Paola, R. A. D. and A. Heller, *Dominical categories: Recursion theory without elements*, Journal of Symbolic Logic **52** (1987), pp. 594–635.

[23] Robinson, E. and G. Rosolini, *Categories of partial maps*, Information and Computation **79** (1988), pp. 95–130.

[24] Timany, A. and B. Jacobs, *Category theory in Coq 8.5* (2015), in the 7th Coq Workshop.
URL http://arxiv.org/abs/1505.06430

[25] Turing, A. M., *On computable numbers, with an application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society **2** (1937), pp. 230–265.

[26] van Doorn, F., J. von Raumer and U. Buchholtz, *Homotopy type theory in Lean*, in: *Proceedings of the 8th International Conference on Interactive Theorem Proving* (2017), pp. 479–495.
URL https://doi.org/10.1007/978-3-319-66107-0_30

[27] Vinogradova, P., "Investigating Structure in Turing Categories," Master's thesis, University of Ottawa (2012).
URL https://ruor.uottawa.ca/bitstream/10393/20505/3/Vinogradova_Polina_2012_thesis.pdf

[28] Vinogradova, P., "Formalizing Abstract Computability: Turing Categories in Coq," Ph.D. thesis, University of Ottawa (2017).
URL https://ruor.uottawa.ca/bitstream/10393/36354/3/Vinogradova_Polina_2017_thesis.pdf

[29] Zammit, V., *A mechanisation of computability theory in HOL*, in: *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics* (1996), pp. 431–446.
URL http://dl.acm.org/citation.cfm?id=646523.694703

[30] Zammit, V., *A proof of the S-m-n theorem in Coq*, Technical report, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK (1997).
URL http://kar.kent.ac.uk/21524