Formalizing Abstract Computability : Turing Categories in Coq

Polina Vinogradova

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science [1]

Department of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

[1] The Ph.D. program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Computer Science

Abstract

The concept of a recursive function has been extensively studied using traditional tools of computability theory. However, with the development of category-theoretic methods it has become possible to study recursion in a more general (abstract) sense. The particular model this thesis is structured around is known as a Turing category. The structure within a Turing category models the notion of partiality as well as recursive computation, and equips us with the tools of category theory to study these concepts. The goal of this work is to build a formal language description of this computation model. Specifically, to use the Coq proof assistant to formulate informal definitions, propositions and proofs pertaining to Turing categories in the underlying formal language of Coq, the Calculus of Co-inductive Constructions (CIC). Furthermore, we have instantiated the more general Turing category formalism with a CIC description of the category which models the language of partial recursive functions exactly.

Acknowledgements

First of all, I would like to thank my supervisors, Dr. Amy Felty and Dr. Philip Scott, for their guidance and financial support, which was absolutely essential for the completion of this dissertation. Without their in-depth understanding of formal logic, knowledge of the requirements and expectations of the process of completing a doctoral dissertation, as well as excellent reference material and problem-solving suggestions, I would have no idea what I am doing. It has been a very positive learning experience working with both of them, and I am grateful they encouraged me to continue on with the completion of this degree when I was considering giving up.

I would also like to thank my supervisors for giving me the opportunity to attend a number of very interesting conferences, where I had many fruitful interactions with other researchers in mine and related fields. I would particularly like to thank Dr. Felty for giving me the idea for the direction of the project undertaken for this dissertation. Working on this project has allowed me to explore a very interesting new perspective on an existing subject.

I would like to express my gratitude to all the fellow students and professors who have participated in the Logic and Foundations of Computing research group — it has been a great experience getting to know everyone interested in logic in Ottawa, and their support and enthusiasm for logic has been a constant motivation for continuing my research. I would like to thank Dr. Pieter Hofstra and Dr. Richard

Blute for organizing so many interesting talks. I also wish to thank Dr. Hofstra for initially suggesting I discuss the possibility of doing doctoral research under Dr. Felty's supervision — I am glad I took this advice.

Finally, I would like to express a huge thank you to my parents and sister, who have been unbelievably understanding and supportive over the time of this long and arduous undertaking. I do not believe that I would have been able to finish my thesis if I had moved out of my family's house and lived on my own. I would also like to thank my friends (some of whom refer to me as 'the one who is writing her thesis') for accepting my standing excuse of 'I have to write my thesis' to get out of anything, and who have consistently believed in the possibility of me one day completing it.

Contents

Chapter 1

Introduction

Traditional computation on the natural numbers, as well as the theory of computability (or recursion) built around it, has historically been the main approach to studying computation in mathematics and computer science, mainly due to its widespread applicability. More recently, there have been several attempts at capturing the essential properties of computation in a broader sense through various mathematical abstractions.

Many concepts of traditional recursion and computation theory have begun to be effectively analyzed categorically. This includes early fundamental work of Elgot on flowchart semantics, and its connections with denotational semantics (cf. M. Arbib and E. Manes [36]), to analysis of Church's theories of lambda calculi (both typed and untyped) in cartesian closed categories and associated higher-order categorical logics ([34]). In a series of works of increasing categorical generality, beginning with Longo and Moggi [35], Di Paola and Heller [40] and culminating in recent work of Cockett and Hofstra [15], we see the beginnings of a new and direct categorical development of the foundations of recursion theory.

Now, 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. For an overview of category theory and how it uses objects, arrows and equations to model mathematical and computer science concepts, see [1], as this outside the scope of this thesis.

The model we have selected to study in this thesis is a category-theoretic approach to exploring more general instances of computational systems, referred to as "Turing Categories" [15]. Turing categories are currently the most general computational model, because they model both total and partial computation and have multiple non-isomorphic objects. 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 formal language description (formalization) of the relevant concepts. A formal language is a set of strings of symbols constrained by rules specific to it. The formal definitions of the model are made up of the symbols representing concepts such as sets or maps, with rules about them; while proofs are essentially sequences of rule applications. The key motivation behind taking this approach is the level of organization, consistency, and guaranteed correctness it provides in working with proofs and definitions for which informal formulations may omit important and interesting details.

Turing category theory can be viewed as an (up until now) informally-presented language that can be used to describe (also informally-presented) formal computation. As computation on a physical computer is a formal procedure, it seems natural to verify that a formal description of it formally fits the selected model. This is the motivating idea and the main objective of this work. The insights into traditional and abstract computability this formalization effort yielded are outlined later in this introduction. There is not a huge amount of work done in this direction of research,

none formalizing specifically a category as an instance of an abstract computational model. For more details on the related work, see Section 5.1.

In particular, our intent was to first use an existing formal language and category theory library written in that language to specify the mathematical definitions found in the framework of the Turing Category abstract computation model, as well as the abstracted versions of other types of structures naturally occurring in the traditional computation model, then formally prove (the more abstract versions of) a number of the results from traditional recursion theory. The proof assistant chosen for this purpose is Coq, with the Calculus of (co)Inductive constructions as the underlying formal language — more details about this decision will be given later in the introduction, as well as Chapters 4 and 5.

In addition to formalizing the categorical concepts, the next key part of this project is formalizing several examples of categories which contain the types of structure relevant to the theory of Turing categories. These examples serve as test cases for the verification of the validity of the formal versions of the definitions and propositions constructed in the formal language (and hence, the related formally proved results about them) as well as to formally study these categories. This is discussed in Chapter 5, with previous work discussed in Sectionand 6.2.

In particular, the main example we focus on in this project is the formalization of traditional computation on the natural numbers and the categorical interpretation of all the structure found therein in order to prove that these indeed conform to the Turing category model formalism. This process is described in Chapter 6, along with presenting a number of interesting observations about the formal approach to integrating partial recursion structure with formal categorical structure.

1.1 Methodology

The following is an outline of the approach we took for completing our formalization project:

(i) Select a model of computation
— Turing categories

(ii) Select formal language and proof assistant software
— Calculus of co-Inductive Constructions and Coq

(iii) Select a library with the tools needed to build the computation model
— Category theory library (due to Timany and Jacobs) [45]

(iv) Formalize categorical structures and definitions relevant to the selected computation model

(v) Formalize motivating examples of each of the formalized categorical structures

Note the formalization referred to in steps (4.) and (5.) is an iterative process. That is, we often need to backtrack and restructure definitions, propositions, and examples in order to build on them. The reason for this is that it may not be immediately clear how, for example, to formally structure a hierarchy using the keywords of a particular feature, or what argument must be given explicitly as opposed to when a proof of existence of the required argument suffices. We discuss the such considerations throughout this thesis.

The following three sections outline concepts and approaches within mathematics and computer science that make up the key considerations for the formalization undertaken within this project, followed by an outline of the contributions of this thesis to the fields of computability and formal logic.

1.2 Traditional Computation

The motivating example for building a more general computational theory is traditional computation. The purpose of traditional computation is to study the natural numbers $\mathbb{N}$ and the computability of functions $\mathbb{N} \to \mathbb{N}$. There are a number of different models of computation that can be used to formally define a universal class of computable functions. According to the Church-Turing thesis, these models all represent the same class of functions, which can be informally described as ones that can be computed according to the intuitive notion of an algorithm. This indicates that studying the set of maps described by just one specific model appears to be sufficient to make conclusions about computability in any sense of the word. In this work, (μ-)recursive functions is the model chosen to formally define computability, and will be specified in the "Formalizing Categorical Examples" chapter.

The collection of all the computable maps can be enumerated by systematically listing all possible functions formed according to the (finite collection of) generating rules of the selected computation model. It follows that every computable function f can be assigned a distinct code c_f in the enumeration. Evaluating the function f at an argument $a \in \mathbb{N}$ can be described using the mapping $(c_f, a) \mapsto f(a)$. The set of all possible computations can then be given by a map $\bullet : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. This map applies the function coded by the first argument to the value of the second argument, and is known as the universal computable function.

It is important to remind the reader that a computation may not necessarily halt on every input. That is, the map $f : \mathbb{N}^n \to \mathbb{N}^n$ corresponding to the input-output pairs of the computation algorithm may not be defined on certain input values (where the algorithm produces no output). This is an important feature of computation known as partiality. It can to be described via special categorical structure. The maps that do halt on every input form a subset of computable maps referred to as *total*, those that do not halt are called *partial*. The domain of a partial computable function

(which is a subset of $\mathbb{N}^n$) is called a recursively enumerable set.

Another interesting property of the standard model of computation is that there are effective encodings of n-ary products of the natural numbers into $\mathbb{N}$ itself, in the form of computable bijections $\mathbb{N}^n \cong \mathbb{N}$, for $n > 0$. This, too, is reflected in the structure of the categorical computational abstraction.

These results, as well as some other facts about traditional computability theory used in this thesis, can be found in [20].

1.3 Choice of Categorical Abstraction Strategy for Traditional Computation

The collection of all recursively enumerable sets, together with the computable maps, forms a category. This category is a subcategory of sets and partial functions, Par, and inherits its structure, along with having the additional structure described above that comes with enumerability of computable maps (for standard category theory results found in this thesis, see [1]). When looking for a more general category to perform computation in, three desirable properties can be highlighted from the above description of traditional computation:

(i) Some kind of categorical structure capturing partiality

(ii) A special object A and an application $\bullet : A \times A \to A$ through which all the computable maps in the category can be factored in a particular way

(iii) Embeddings (or encodings) of every other object in the category into A

A well-known categorical model for the simply typed lambda calculus computation formalism is a Cartesian closed category (CCC) [34]. CCCs provide a method for

interpreting maps in the category via the function space objects of that category and the *eval* map, but, in general, do not contain a single "universal" such object suitable for factoring all maps. Furthermore, the concept of partiality is not specifically reflected in the structure of CCCs.

A more recent categorical framework which incorporates these three features is called a Turing category. Turing categories were first introduced in the 2008 paper "Introduction to Turing Categories", by J. R. B. Cockett and P. J. W. Hofstra [15]. Partiality in such a category is handled by means of idempotents: we may represent the domain of a partial map in the form of a specific type of idempotent. Then, the special object A and the application $\bullet$ generalize the idea of a universal computable function. Finally, the condition that every object in such a category is a retract of A generalizes the fact that all recursively enumerable sets of powers of $\mathbb{N}$ are isomorphic to recursively enumerable subsets of $\mathbb{N}$.

The motivating example for defining Turing categories, as described above, has a lot of additional structure due to the very special nature of the natural numbers and computation on them. Part of the motivation behind studying the Turing category abstraction is exploring the process of bridging the gap between this specific example and the completely general setting with the minimal structure necessary to represent what it means to perform computation, by adding structure to a Turing category in a controlled way.

Describing existing computational models in terms of Turing structure showcases its versatility and allows us to get away from a set-theoretic view of computation while still maintaining a coherent notion of partiality (including domains and ranges). This approach ties together all the necessary features of computation as well as related concepts found in traditional recursion theory, modeling closely the way that they interact in the computation model. For this reason, the Turing category model appears to be most useful as a way to apply category-theoretic machinery to studying computation.

1.4 Choice of Formalism and Category Theory Library

The other key ingredient of this project is the choice of formalism used to develop the formal logic description of the categorical abstraction. The formal language selected for this purpose is the calculus of (co)inductive constructions (CIC). For reference on CIC [28]. This formal language is implemented using the proof assistant software "Coq" [22, 4], which uses the rules of formal logic within CIC to verify (that is, to type-check) the user-entered proofs and definition structures.

CIC is an extension of the calculus of constructions, modified to include inductive types as well as co-induction. The calculus of constructions was developed by Thierry Coquand and is a higher order typed lambda calculus. At this point, the reader should be reminded that the set of rules for computation in the formal language CIC is an extension of the set of rules for computing in the simply typed lambda calculus (STLC). The type-formation rules in CIC allow for the definition of a very large variety of types (while still maintaining language consistency) — a feature very important for an intuitive and precise formalization of existing category-theoretic concepts.

CIC is the underlying formal language implemented in the extensively developed and widely used proof assistant software Coq. The Coq type checking system verifies and certifies user-defined terms expressed in the CIC language as well as assists in the search for formal proofs of (formally expressed in CIC) mathematical and logical assertions about these well-typed terms. That is, it provides an environment which tracks the current goal (of type Prop when building a proof), as well as the terms and hypotheses in the scope of the current goal, and allows the user to manipulate the goal by using tactics (including Coq's proof automation) to arrive at a constructive proof expressible in CIC. The end result is a lambda term in CIC representing a proof

(again, verified and certified by Coq).

Coq is best known as the proof assistant used in proving results in the fields of algorithms (the Four Colour Theorem [24], Disjoint-Set data structure [9]) as well as group theory (the Feit-Thompson theorem, [25]) and compiler correctness (CompCert), [5] (more on these in the Formalization and Coq chapter). Compared to other formalization software, Coq has more extensive documentation available online, a wide selection of libraries for a variety of purposes, and also a comprehensive textbook on programming using the software and its key features [43]. Furthermore, there have been a number of user-friendly interfaces designed for the Coq proof assistant. For these reasons, we have decided to use Coq as the formalization tool for this project.

One of the attractive features of the built-in Coq library are type classes, which provide a way to bundle a collection of related formalisms into easy-to-manipulate structures (these will be elabourated on in Chapter 5). There are a number of libraries for formal mathematics written for the Coq proof assistant which make use of the type class feature. The instantiation typically requires providing the key components of a category: objects, maps, composition, identities (with corresponding equations), and associativity. The formalization of every other categorical concept included in a particular library is constructed using these terms and equations. The specific organization, shortcuts, etc. of the library are unique to the individual formalization effort.

The library selected for the purpose of this project is due to Amin Timany and Bart Jacobs [45]. It is a library developed using a mainstream version of Coq. The version of the Coq proof assistant software as well as the development environment used (CoqIDE) for this formalization project is 8.5beta2. This is the Coq version in which the category theory library used for my formalization project is developed, and this library is not compatible with newer versions of the proof assistant on a Windows system. The project is built on a standard 64-bit Windows 8.1 system with

all required updates installed.

The library used for this formalization project appears to be currently the most complete, easy-to-use, well-documented and clearly organized such library. It contains all the features necessary for formalizing Turing Categories as well as the concepts necessary to formalize structure found in traditional computation (that can be added to a Turing category as needed). These are the reasons behind selecting this particular library (for more details, see Chapter 5).

1.5 Plan of the Thesis

The first two chapters give the relevant category theory and formal language background, respectively, necessary to discuss the formalization undertaken in this project. A background knowledge of basic concepts in each area is assumed.

Chapter 2 covers the approach to categorically expressing the concept of partiality. It then goes on to give an overview of Turing categories, developed by Hofstra and Cockett in [15], as well as certain results about them, underlying computational structure, and examples. Note that this chapter and the "Ranges in Turing Categories" (Chapter 3) correspond very closely to the similarly titled chapters in my M.Sc. thesis [47].

In Chapter 4, the calculus of (co)inductive constructions is described in detail. In addition, certain features specific to Coq are outlined, the knowledge of which is needed to understand and parse the category theory library selected for the abstract computability formalization. The following chapter presents some other existing category theory formalization library options that were candidates for use as the backbone of the formalization, as well as existing formalization efforts of certain concepts prerequisite for Turing categories.

Chapter 3 contains original research completed as part of my M.Sc. thesis [47]. In this chapter, the concept of range of a function is generalized: the partial identity

map on an object in a given category, defined only on the range of some function, is abstracted by a special type of idempotent. The main problem addressed is finding necessary and sufficient conditions for a Turing category to have ranges, and to understand what they look like.

The next two chapters correspond to the files of the formalization code developed in this work. The formalization code totals 4,866 lines of code (excluding experimental code left as comments at the end of each of the files). The up-to-date instructions on the use of these files are found in the ReadMe file accompanying them on the GitHub page. This page is found at

"https://github.com/polinavino/Turing-Category-Formalization"

Chapter 5, which focuses on the **Restriction.v, Turing.v, PCA.v, CompA.v**, and **Range.v** files, goes into detail explaining the approach to formalizing the background categorical structure (including partiality and cartesian restriction structure) found in the file, then the formalization of Turing structure itself. Our formalization of a computational model known as a partial combinatory algebra (PCA), which is related to Turing structure is also discussed in this category. We then go on to describe the formalisms we have defined to express for the subcategory of a Turing category corresponding to the PCA structure found therein. This chapter also discusses our approach to the formalization of range structure (as described in Chapter 3).

Chapter 6 focuses on the **Restriction.v** and **CompN_Cat.v** files. In this chapter, we discuss the formalization of examples illustrating the categorical structure defined in Chapter 5. We begin by formalizing the category of sets and partial functions, Par, to illustrate the use of Cartesian restriction and range category structure. We then formalize, as a subcategory of Par, the motivating example for Turing categories — the category of sets of the form $\mathbb{N}^n$ and partial recursive maps between them.

1.6 Contributions

Describing an abstract setting in terms of formal logic yields an interesting new perspective on the subject, and the resulting formalization also serves to organize the results on abstract computability in a structured way to facilitate or potentially even automate future research on the topic. In particular, the key contributions we have made as part of research done for this thesis are as follows:

(i) Formalized the categorical structure (and relevant proofs of propositions) prerequisite to considering the existence of Turing structure in a given category — that is, cartesian restriction structure

(ii) Formalized Turing structure as well as existing results about Turing structure, including the abstracted versions of certain key recursion theory results

(iii) Formalized another type of structure that may be found in Turing categories under certain conditions — ranges of maps, and the way this structure is expected to interact with Turing structure when both are present in a category

(iv) Formalized the category of sets and partial maps, along with all the structure (in (i) and (iii)) as it is defined in this category

(v) Formalized the motivating example of traditional computability

By completing this formalization project, we:

(vi) Have conducted a thorough study of the expressibility of the concept partiality in a formal language both in an abstract sense and a concrete set-theoretic definition

(vii) Explored the limitations and challenges of describing a more abstract computational setting by means of a formal language

(viii) Obtained new insights into the specifics of structure within a category modeling traditional computation, both directly and as an instance of a more abstract Turing category model

(ix) Built a consistent, hierarchically structured system of classes of categories that defines exactly the relationships between these classes and the structures defined within each (to compliment the ad-hoc informal definitions)

(x) Gained insight into the application of Coq in the domain of computability and category theory

Chapter 2

Turing Category Background

2.1 Restriction Categories

In order to explain Turing structure, as well as what sort of category it can exist in and how exactly it abstracts computation, the concept of partiality must first be introduced. Partiality of a function, informally, refers to the idea that a map $f : A \to B$ may not be defined on the whole of A. That is, if f is, say, a map between two sets A and B, f may not be defined on some elements of A. The notion of partiality, expressed via idempotents, has been developed to represent the 'domain' of a function. In this and subsequent chapters, the definitions of common categorical concepts will not be stated explicitly — for reference on omitted definitions see [1].

Partiality is a key concept in abstract computability because it is responsible for a number of features of traditional recursive functions. Recursive functions are maps between sets of the form $\mathbb{N}^n, n \in \mathbb{N}$. These functions (computations) do not always produce an output on every input because the corresponding algorithm may not ever finish computing on the given input (this is a well-known result in computability [20]). Thus, we can treat these maps as maps in the larger category Par of sets and partial maps.

We begin axiomatizing partiality by studying its structure in the case of Par. We can express the domain of a map in this category as follows:

Example 2.1.1 In Par, define the idempotent representing the domain $\overline{f}$ of a map $f : X \to Y$:

$$\overline{f}(x) = \begin{cases} x & \text{if } f(x) \downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

In this example, and subsequently, $f(x) \downarrow$ means that f is defined at x, and $\uparrow$ means it is undefined. The resulting map is equal to the identity on X wherever $f(x)$ is defined, and is undefined otherwise. For a discussion of the formalization of this example, see Section 6.1.

The above definition of $\overline{(-)}$ can be generalized and axiomatized for an arbitrary category. Partiality can be introduced into a category in the following way, making the category into a restriction category [16]:

Definition 2.1.2 A **restriction category** is a category C endowed with a combinator $\overline{(-)}$, sending $f : A \to B$ to $\overline{f} : A \to A$, such that the following axioms hold:

[**R.1**] $f\overline{f} = f$

[**R.2**] $\overline{f}\overline{g} = \overline{g}\overline{f}$ whenever $\mathrm{dom}(f) = \mathrm{dom}(g)$

[**R.3**] $\overline{g\overline{f}} = \overline{g}\overline{f}$ whenever $\mathrm{dom}(f) = \mathrm{dom}(g)$

[**R.4**] $\overline{g}f = f\overline{gf}$ whenever $\mathrm{cod}(f) = \mathrm{dom}(g)$

The intuition behind these four rules comes from attempting to observe and represent the way composition of maps should behave with respect to 'domains' and the way composing maps will affect the domain of definition of the resulting map. For example, R.1 says that pre-composing with the domain of a map does not change (or restrict) the original map. The category of sets and partial functions, Par, can be viewed as a general motivating example for describing partiality in this way.

In the above example, The map $\overline{f}$ corresponds to the domain $S \subseteq X$ of the map f in the sense that it is defined (and coincides with 1_X) exactly on that subset of X. It is straightforward to check that this definition satisfies the conditions in Definition 2.1.2. Note that in this category the map $\overline{f}$ is actually a partial identity on X. Furthermore, with this example it is easy to see that a restriction combinator is not necessarily unique in a category — another possibility for Par is simply $\overline{f} = 1_X$, which also satisfies all the axioms in a trivial way.

This is also the time to emphasize the importance of functional extensionality. In Set, this amounts to: $f = g$ whenever for all x, $f(x) = g(x)$. Without a generalized version of this rule governing the equality comparison of two partial set maps,

$$f = g \text{ whenever } (f(x)\downarrow \Leftrightarrow g(x)\downarrow \text{ , and } f(x) = g(x) \text{ if both } \downarrow)$$

it would be impossible to prove, for example, that the restriction combinator, as defined above, indeed satisfies the required four rules stated in the definition. In general, the majority of equality proofs between functions, partial functions (including both computable and no*n*-computable ones), and relations between sets rely on judgments of comparisons of the maps in question evaluated at each individual element in the given set. Functional extensionality, as well as this more general extensional comparison of partial maps, is discussed in more detail with respect to implementation in a formal system in Section 4.3.

The concept of a restriction combinator prompts a generalized definition of a total function: a map $f : X \to Y$ is total whenever $\overline{f} = 1_X$. An example of a total map in a restriction category C comes up when studying a special pair of maps, called an embedding-retraction pair, defined as follows:

Definition 2.1.3 Suppose C is a category with objects X and A. X is said to be a **retract** of A if there exists a pair of maps, $m_X : X \to A$ and $r_X : A \to X$, such that $r_X m_X = 1_X$. The pair (m_X, r_X) is called an **embedding-retraction pair**, also

denoted

$$m_X : X \lhd A : r_X \text{ or } (m_X, r_X) : X \lhd A$$

.

The above definition will later play an important role in characterizing maps in a Turing category. Note that in the above definition, the map r_X is an epimorphism, while m_X must be total:

$$\overline{m_X} = r_X m_X \overline{m_X} = r_X m_X = 1_X$$

Furthermore, it is possible to define a subcategory of total maps of C:

Definition 2.1.4 Suppose C is a restriction category. Then, the **total subcategory of** C, denoted Tot(C), is the subcategory of C consisting of all the objects of C, and only the total maps of C.

This total subcategory is an instance of a wide subcategory of the larger category. Recall that a wide subcategory is a subcategory that contains all the objects of the larger category and only some of the morphisms. Another important structure that naturally exists in a restriction category C is a partial ordering $\leq$ on hom-sets, defined as follows:

$$f \leq g \Rightarrow g\overline{f} = f$$

For example, in the case of sets and partial functions, $f \leq g$ means that the domain of f is a subset of the domain of g, and that $f(x) = g(x)$ for all x in the domain of f.

Often, there is a minimal map with respect to this ordering, denoted $0_{A,B} : A \to B, A, B \in \mathsf{C}$, such that for all $f : A \to B, 0_{A,B} \leq f$. In the case of sets and partial functions, this amounts to the map $A \to B$ which is undefined for all $a \in A$. However, the existence of such a map is not guaranteed.

Splitting Idempotents. Because idempotents are essential for reasoning about partiality, it is necessary to give an explicit definition of what an idempotent is, and what it means for one to split [21]. The following definition combines traditional idempotent concepts with restriction structure, defined in [16].

Definition 2.1.5 Let C be a category, and a map $e : X \to X$ in C. Then

(i) e is said to be an **idempotent** when $ee = e$

(ii) e is said to **split** when there exists an object $Y \in \mathsf{C}$, as well as maps $m : Y \to X$ and $r : X \to Y$ such that $e = mr$, and $rm = 1_Y$.

(iii) idempotents e, e' are said to be **equivalent** when they have isomorphic splittings

(iv) e is a **restriction idempotent** when $\overline{e} = e$.

Note that all restriction maps $\overline{f} : A \to A$ are restriction idempotents, and $\overline{f} = f$ implies $ff = f\overline{f} = f$. Furthermore, each restriction idempotent e is the restriction of some map (such as e itself).

From this point on, it will sometimes be assumed that all idempotents in a category C split. Alternatively, Split(C), referred to as the Karoubi envelope of C, will denote the category obtained from C by formally splitting all idempotents [21, 34].

Definition 2.1.6 Given a category Par, we define the **Karoubi envelope** of C, denoted Split(C), as follows:

The objects in this category are pairs of the form (A, e) where $A \in \mathsf{C}$ and $e : A \to A$ is an idempotent of C, and whose morphisms are triples of the form

$$(e, f, e') : (A, e) \to (A', e')$$

where $f : A \to A'$ is a morphism of C satisfying $e'f = f = fe$.

In the category Split(C), the splitting of an idempotent e may now be given by the object (A, e), with an embedding $(e, e, 1)$ and retraction (e, e, e). As with any idempotent splitting, the object (A, e) is unique up to isomorphism.

The following example illustrates the above structure in the category describing traditional computation.

Example 2.1.7 Let $\mathbb{N}$ denote the one-object category of natural numbers and partial recursive functions. Here, restriction structure is inherited from Par, and restriction idempotents in this category correspond to recursively enumerable subsets of $\mathbb{N}$ (which are exactly the the domains of partial recursive functions). For a general idempotent $e : \mathbb{N} \to \mathbb{N}$, the image of e is again a recursively enumerable set. Thus, the images of idempotents can be represented by recursively enumerable subsets, and the objects of Split($\mathbb{N}$) may be taken to be the r.e. subsets.

Open Maps. The category of topological spaces and partial continuous functions with open domains is another noteworthy example of a restriction category. There is a special class of maps in this category, namely the *open maps*: a continuous function $f : X \to Y$ is called open when the direct image of an open set in X under f is an open set in Y [32]. The concept of open maps has been previously studied in the categorical setting [30].

In [13], the authors showed how the concept of openness can be defined in a general restriction category. Let $\mathcal{O}(A)$ denote the poset of restriction idempotents of an object A in a restriction category C. Then, for $f : A \to B$, let $f^* : \mathcal{O}(B) \to \mathcal{O}(A)$ denote the "inverse image" function such that for any $e \in \mathcal{O}(B)$, $f^*(e) = \overline{ef} \leq \overline{f}$.

Furthermore, composing restriction idempotents is denoted by $ee' = e \wedge e'$. This $\wedge$ operation is actually the meet in the poset $\mathcal{O}(A)$; a definition which extends this to more general types of maps will be given in the "Equality" chapter.

Definition 2.1.8 In a restriction category, a map $f : A \to B$ is **open** if and only if there is a poset morphism $\exists_f : \mathcal{O}(A) \to \mathcal{O}(B)$ such that
[O1] $\exists_f(f^*(e')) \leq e'$ for all $e' \in \mathcal{O}(B)$
[O2] $e \wedge f^*(e') \leq f^*(\exists_f(e) \wedge e')$ for all $e \in \mathcal{O}(A), e' \in \mathcal{O}(B)$

[O3] $e' \wedge \exists_f(e) \leq \exists_f(f^*(e') \wedge e)$ for all $e \in \mathcal{O}(A), e' \in \mathcal{O}(B)$

In the case of Par, we can express both f^* and $\exists_f$ in terms subsets of A and B in Par to which the idempotents e and e' correspond,

$$f^*(e) = \{x \in A : f(x) \in e\} \qquad \exists_f(e') = \{f(x) : x \in e'\}$$

Note that generally, in a restriction category, the map f^* actually lets us recover the restriction of f, as $f^*(e) = \overline{ef}$, and taking $e = 1$ gives the result $f^*(1) = \overline{f}$. The map $\exists_f$ is almost left adjoint to the f^* map; however, since f may be partial this is only true when we restrict $\exists_f$ to the domain of f. The O1 axiom gives the counit of the almost-adjunction. The next two axioms make sure the two maps behave correctly with respect to composition with idempotents.

With this open map definition, it turns out that in any restriction category, all restriction idempotents are open maps, and moreover, the collection of open maps is closed under composition [13].

Lemma 2.1.9 Suppose $f : A \to B$ and $g : B \to C$ are open maps in a cartesian restriction category. Then the composition gf is also an open map.

Now, the upcoming definitions provide the setting for an even more specific class of categories that Turing categories are a part of — cartesian restriction categories, described in [15].

2.2 Cartesian Restriction Categories.

Definition 2.2.1

(i) Let 1 be an object in a restriction category C. The object 1 is **restriction terminal** when for any object $A \in \mathsf{C}$, there exists a unique total map $!_A : A \to 1$, such that for every $f : A \to B$, $!_B f = !_A \overline{f}$.

(ii) A **restriction product** of two objects $A, B \in \mathsf{C}$ is an object $A \times B$, together with total projections $\pi_A : A \times B \to A, \pi_B : A \times B \to B$ such that for each pair of maps $f : C \to A, g : C \to B$ there exists a unique map $\langle f, g\rangle : C \to A \times B$ such that $\pi_A\langle f, g\rangle \leq f, \pi_B\langle f, g\rangle \leq g$, and $\overline{\langle f, g\rangle} = \overline{f}\overline{g}$.

This definition implies that a restriction terminal object in C is a genuine terminal object in Tot(C), and a restriction product $A \times B$ is an actual product in Tot(C). This follows because when we consider the restriction-terminal object and restriction product diagrams in the total map subcategory, they must now commute exactly instead as inequalities.

Definition 2.2.2 A restriction category C is called **cartesian** if it has all binary (partial) products as well as a (restriction) terminal object.

Consider the following examples.

Example 2.2.3 (i) In Par, the partial terminal object would be the same as the terminal object in Set, which is the one-element set $\{*\}$. The partial products of Par would be genuine products in Set.

(ii) In the category of r.e. subsets of the natural numbers and partial recursive maps between them, the partial terminal object, again, is a one-element set, and the partial products are pairs of ordered pairs (triples, etc.). That is, this category inherits cartesian restriction structure from Par.

2.3 Turing Category Structure

We now turn to the key definition used in this thesis, namely that of a Turing category (as stated in [15]). Formalizing the following lemmas and definitions are the backbone of this project, and are the basis of results used in the subsequent chapters.

Definition 2.3.1 Let C be a cartesian restriction category.

(i) Given a morphism $\tau_{X,Y} : A \times X \to Y$, a morphism $f : Z \times X \to Y$ is said to **admit a $\tau_{X,Y}$-index** h when there exists a total map $h : Z \to A$ for which the following diagram commutes:

$$\begin{array}{ccc} A \times X & \xrightarrow{\tau_{X,Y}} & Y \\ {\scriptstyle h\times 1_X}\uparrow & \nearrow_{f} & \\ Z \times X & & \end{array}$$

(ii) The morphism $\tau_{X,Y}$ is called a **universal application** when every $f : Z \times X \to Y$ admits a $\tau_{X,Y}$-index.

(iii) A **Turing object** in C is an object A such that for each $X, Y \in \mathsf{C}$, there is a universal application $\tau_{X,Y} : A \times X \to Y$.

(iv) The category C is called a **Turing category** if it possesses a Turing object.

It is important to note that Turing categories are different from cartesian closed (CC) categories. CC categories have a terminal object, all finite products and all exponentials Y^X. This amounts to the natural bijection $Hom(Z \times X, Y) \cong Hom(Z, Y^X)$. The diagram depicting this is somewhat similar to that used in the definition of Turing categories:

$$\begin{array}{ccc} Y^X \times X & \xrightarrow{ev} & Y \\ {\scriptstyle \tilde{f}\times 1_Y}\uparrow & \nearrow_{f} & \\ Z \times X & & \end{array}$$

One difference is that in a Turing category, $\tilde{f}$ need not be unique. The preceding definition pertains to the defining property of a Turing category — the presence of a Turing object. Now, the following definition will outline the key structure found in a Turing category.

Definition 2.3.2 Let C be a Turing category and A an object of C.

(i) A family of maps $\tau = \{\tau_{X,Y} : A \times X \to Y | X, Y \in \mathsf{C}\}$ is called an **applicative family** for A.

(ii) An applicative family τ for A is called **universal** when each $\tau_{X,Y}$ is universal. In this case, τ is also called a **Turing structure** on A.

(iii) The pair (A, τ), where τ is universal for A is called a **Turing structure** on C, and τ, also denoted $\bullet$, a **Turing morphism**.

Suppose $\tau_{X,Y} : A \times X \to Y$ is a universal application in C, and $f : Z \times X \to Y$ is a map. Since C has all products, X may itself be an arbitrary product, we will consider the case when $Z = 1$. In this case, the index for the map f, denoted $c_f : 1 \to A$, is called a code for f. The isomorphism $1 \times X \to X$ will often be suppressed in this and the following chapters.

To characterize the structure of maps and Turing objects in a Turing category, the following lemmas are useful.

Lemma 2.3.3 Suppose C is a Turing category with Turing object A. Then every $X \in \mathsf{C}$ is a retract of A.

The embedding-retraction pair $X \lhd A$ is given by $(\widetilde{\pi_X}, \tau_{1,X}\pi_A^{-1})$, as in the following diagram:

$$\begin{array}{ccc} A \times 1 & \xrightarrow{\tau_{1,X}} & X \\ {\scriptstyle \widetilde{\pi_X}\times 1}\uparrow & \nearrow_{\pi_X} & \\ X \times 1 & & \end{array}$$

This implies that $\tau_{A,A}$ is a generic universal application in a Turing category. Observe that an arbitrary map $f : X \to Y$ factors through $\tau_{A,A}$, via an embedding $m_X : X \to A$ and a retraction $r_Y : A \to Y$:

$$\begin{array}{ccc} A \times A & \xrightarrow{\tau_{A,A}} & A \\ {\scriptstyle c_f \times m_X}\uparrow & & \downarrow{\scriptstyle r_Y} \\ 1 \times X & \xrightarrow{f} & Y \end{array}$$

Next, let us consider the possibility of multiple Turing objects in a Turing category. As proved in [15], section 3.3, lemma 3.5, it is indeed possible for there to exist

more than one Turing object:

Lemma 2.3.4 Suppose C is a Turing category with Turing object A. Then $A' \in \mathsf{C}$ is a Turing object of C if and only if A is a retract of A'.

Note that even though both Turing objects A and A' are retracts of each other, they need not be isomorphic.

These are important results about Turing categories and are part of the formalization project undertaken for this thesis. Now, let us consider the motivating example of recursive computation on the natural numbers.

Example 2.3.5 The standard model for computation is the Turing category of finite powers of the natural numbers — $\{1, \mathbb{N}, \mathbb{N}^2, ...\}$, together with the (enumerable) set of all computable functions $\mathbb{N}^n \to \mathbb{N}^m$. A Turing object in this category is $\mathbb{N}$, and there exist embedding - retraction pairs for $\mathbb{N}^n, n \geq 1$ which are in fact isomorphisms — this is not generally true in all Turing categories. The $\mathbb{N}^2 \to \mathbb{N}$ isomorphism may be defined by

$$(a, b) \mapsto (a + b)(a + b + 1)/2 + a$$

The application $\tau_{\mathbb{N},\mathbb{N}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is defined as follows: for any $n, m \in \mathbb{N}$,

$$n \cdot m = \phi_n(m),$$

where ϕ_n is the computable function with index n. This type of notation is known as Roger's notation [20]. Also, here, we use notation $(-)\cdot(-)$ for the application $\bullet\langle -, -\rangle$. The index comes from a Gödel numbering framework of all computable functions $f : \mathbb{N} \to \mathbb{N}$.

It is important to note that the category of powers of $\mathbb{N}$ and (partial) maps $f : \mathbb{N}^n \to \mathbb{N}^m$ computable by partial recursion is a subcategory of Par, with the same restriction terminal object and same restriction products. Furthermore, equality between maps in $\mathbb{N}$, as it is inherited from Par, is the partial map version of functional extensionality defined earlier.

Describing a Turing category in terms of factorizations of all maps $Z \times X \to Y$ directly through the applicative map may not be the most convenient way to study Turing structure in all cases. Using the results from the lemmas presented above, it was shown in [15] that a Turing category may be equivalently characterized in terms of just the applicative map itself and embedding-retraction pairs for all other objects into the Turing object.

Theorem 2.3.6 Suppose C is a restriction category. Then, describing C as a Turing category with Turing object A and Turing morphism is equivalent to C satisfying the following conditions:

(i) there exists an object $A \in \mathsf{C}$ and a map $\bullet : A \times A \to A$ such that this map is a Turing morphism, and

(ii) every object $B \in \mathsf{C}$ is a retract of A

The above example demonstrates that a category of partial recursive maps indeed conforms to the Turing structure model. This would suggest that it is also possible to establish more abstract versions of traditional recursion theory theorems, such as the m-completeness of the halting set. Recall that in traditional computability theory, the halting set is $\{(x, y) : x \bullet y \downarrow\}$. That is, all pairs of natural numbers (x, y) such that the computation corresponding to the number x halts on input y.

In the case of Turing categories, the role of a set will be played by a restriction idempotent (domain), so that the halting set will correspond to the restriction of the applicative map $\bullet : A \times A \to A$. Here the first A represents the function space, and the second is the object on which they are computed via the application. The following definition specifies what it means for a domain to be m-complete in a restriction category:

Definition 2.3.7 Given two restriction idempotents, e on object X and e' on object Y, $e \leq_m e'$, that is, e **many-to-one reduces**, or m**-reduces**, to e' whenever there is

a total map $f : X \to Y$ such that $e = \overline{e'f}$. The domain $\overline{f}$ of a map f is said to be m-**complete** if every other domain m-reduces to it.

To state the Turing category version of the m-completeness result,

Lemma 2.3.8 In a Turing category with applicative map $\bullet$, the domain $\overline{\bullet}$ is m-complete.

For proof of the result, see [15], Corollary 3.9. The above lemma and theorem are some of the results we have formalized, discussed in Section 5.4.

The structure described in the following section will now give the reader a better idea of the nature of an applicative map in a Turing category, and how it relates to the concept of computation.

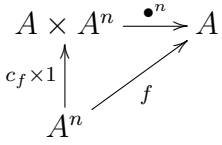
2.4 Partial Combinatory Algebras

Let C be a cartesian restriction category. An applicative system $\mathbb{A} = (A, \bullet)$ in C consists of an object A and a morphism $\bullet : A \times A \to A$, called an application. Now, define $\bullet^n : A \times A^n \to A$ inductively for $n = 2, 3, \ldots$, by $\bullet^n = \bullet(1 \times \bullet^{n-1})$ and $\bullet^1 : A \times A \to A$. Finally, in order for $\bullet^0$ to enjoy the interesting properties of the other maps in the $\bullet^n$ family, it is defined $\bullet^0 = \bullet\Delta$, where $\Delta : A \to A \times A$ is the diagonal map. The following definition, found in [15], describes partial combinatory algebras in terms of its maps and objects.

Definition 2.4.1 Suppose C is a cartesian restriction category, and $A \in \mathsf{C}$. Suppose also that $\bullet : A \times A \to A$ is a map in C, referred to as the **application**.

(i) Consider the smallest cartesian restriction subcategory of C containing every total map $1 \to A$, as well as the application. Any morphism $f : A^n \to A^m$ in this subcategory is called a **polynomial** morphism (or map).

(ii) A morphism $f : A^n \to A$ is said to be $\mathbb{A}$-**computable** when it factors through $\bullet^n$ via some code $c_f : 1 \to A$. A morphism $A^n \to A^m$ is $\mathbb{A}$-computable when each of its m components $A^n \to A$ is.



(iii) When every polynomial morphism $A^n \to A^m$ is $\mathbb{A}$-computable, the applicative system $\mathbb{A}$ is said to be **combinatory complete**, and is referred to as a **partial combinatory algebra**, abbreviated PCA.

Suppose $\bullet : A \times A \to A$ is an application in C. Write $\mathbb{A} = (A, \bullet)$, and let Comp($\mathbb{A}$) denote the category where

The objects of Comp($\mathbb{A}$) are $\{1, A, A^2, \dots\}$
The morphisms of Comp($\mathbb{A}$) are (all) polynomial morphisms.

The following result, also found in [15], describes the connection between Turing structure and combinatory completeness.

Lemma 2.4.2 When the system $\mathbb{A}$ is combinatory complete, Comp($\mathbb{A}$) is a Turing category, with Turing object A. Furthermore, when every object in C is a retract of A, A is also a Turing object in C, with $\tau_{A,A} = \bullet$.

Another related category of interest is the full one-object subcategory A of Comp($\mathbb{A}$), $A \in \mathsf{A}$, with all $\mathbb{A}$-computable maps $f : A \to A$. This is not, however, a Turing category, since it does not contain a terminal object. To summarize, a Turing category C with Turing object A gives rise to three related categories based on PCA $\mathbb{A}$, which all have similar underlying structure:

(i) A,

and the following, which are Turing categories

(ii) Split($\mathbb{A}$)

(iii) Comp($\mathbb{A}$), and

(iv) Split(Comp($\mathbb{A}$))

The relationship between these categories can be described by:

Lemma 2.4.3 These categories embed as follows:

$$\mathsf{A} \hookrightarrow \text{ Comp}(\mathbb{A}) \hookrightarrow \mathsf{C} \hookrightarrow \text{ Split}(\text{Comp}(\mathbb{A}))$$

The formalization of the above lemma is described in Section 5.5.3. Alternatively, a PCA can be identified by the presence of specific elements in A [15]. Here, the notation ab, $a, b, \in A$ is shorthand for $a \cdot b \in A$, and $abc = (ab)c, c \in A$, whereas $ab\downarrow$ denotes that ab is defined. The $\cong$ symbol denotes Kleene equality, which, in the case of Turing categories, takes the form:

Definition 2.4.4 Suppose C is a category, and $f, g : X \to Y$ are two maps. **Kleene equality** $f \cong g$ is defined as follows :

For any point $x : 1 \to X$, we define $f(x) \cong g(x)$ to mean:
$f(x)$ is defined iff $g(x)$ is defined, and in that case $f(x) = g(x)$. We say $f \cong g$ if for all points $x : 1 \to X$, $f(x) \cong g(x)$.

Kleene equality can be viewed as a generalization of extensional equality to cases when the domain of a (partial) map is smaller than the whole source object, and ensures maps can only be considered equal if they have the same domain. Turing categories where Kleene equality holds are called 'extensional'. Now, the following lemma makes the connection between a combinatory algebra and an applicative structure.

Lemma 2.4.5 A partial applicative structure $(A, \bullet)$ is combinatory complete whenever it has elements k and s such that for all $a, b \in A$:

(i) $kab \cong a$

(ii) $sa\downarrow, sab\downarrow$, and $sabc \cong ac(bc)$

In a combinatory complete structure, elements k and s, as well as all elements that can be built up from these and other elements of A by using the application $\bullet$, are called **combinators**.

The above lemma is in set-theoretic notation, but it may also be interpreted diagramatically. Combinators are usually considered as elements of the underlying PCA set. However, in a Turing category, this may be too specific, as the object A in a PCA is not necessarily a set (in Par).

To completely integrate the concept of PCA combinators into an arbitrary Turing category, they must be regarded as the total point maps into a Turing object, $a : 1 \to A$. That is, they are exactly the codes for the maps $\mathrm{Comp}(\mathbb{A})$. This is also the reason behind using Kleene equality instead of equality in Definition 2.4.5.

In the general categorical case, the application $a\cdot b$ is shorthand for $\bullet\langle a, b\rangle : 1 \to A$, where $a, b : 1 \to A$ are codes. A map $f : A \to A$ with a code c_f may be denoted by $c_f \cdot - = \bullet\langle c_f !_A, 1_A\rangle$. Observe that the combinators k, s indeed represent maps in a Turing category with Turing object A:
k is a code for $\pi_0 : A \times A \to A$, whereas s is a code for the composition

$$A \times A \times A \xrightarrow{\langle \pi_{13}, \pi_{23}\rangle} (A \times A) \times (A \times A) \xrightarrow{\bullet\times\bullet} A \times A \xrightarrow{\bullet} A$$

Many different computations can be built up with just those combinators. The combinator k is a code for a projection onto the first coordinate. The following are a few other combinator computations:

$$\begin{aligned} ((sk)k)x &\cong (kx)(kx) \\ &\cong x \end{aligned}$$

$$\begin{aligned} k^*xy &= ((sk)x)y \\ &\cong ky(xy) \\ &\cong y, \end{aligned}$$

Here, k^*, often denoted $\lambda xy.y$, is the combinator for the projection onto the second coordinate. Certain properties of Turing categories, such as existence of range maps or coproducts, will actually enforce the existence of other particular combinators in the underlying PCA.

A PCA structure may be regarded as a model of a certain theory called combinatory logic. There is a close connection between combinatory logic and lambda calculus. For example, the combinator k is present in lambda calculus in the form $\lambda ab.a$. All the lambda calculus computations are, in turn, expressible through the application within a PCA. A proof of this statement may be found in [27]. The following are some further examples of PCA's:

Example 2.4.6 (i) Partial Recursive Computation with an Oracle. Suppose A is any subset of $\mathbb{N}$. Then the characteristic function χ_A of A (which is necessarily total) is defined by:

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise.} \end{cases}$$

Now, consider the category made up of the same objects as Comp($\mathbb{N}$), but whose collection of maps is the closure (under composition) of the set of maps of Comp($\mathbb{N}$) and the map χ_A. Computation using this new set of functions is referred to as computation with an oracle, and the functions are called A-computable. Such a set of functions is again enumerable $(\mathbb{N}, \bullet_A)$, regardless of whether χ_A is in fact recursive. This type of computation with an oracle is described in [20].

Thus, an applicative map, denoted $\bullet_A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, can be defined for this

category similar to the standard computation case, with $n \cdot_A m = \phi_n(m)$. This universal map, as well as the maps required to demonstrate the S_n^m parametrization theorem for A-computability, can be used to build Turing structure. In particular, these have been proved to be A-computable (see [23], pp.151-152), and thus the resulting Turing structure is indeed contained in the category of A-computable maps. In the case when A is not recursive, this category is distinct from the standard computation model, as the sets of computable functions in the two categories are clearly distinct.

(ii) Reflexive Structures in a Cartesian Closed Category. An object A is said to be *reflexive* whenever $A^A \lhd A$. For a reflexive A, the application can be defined as the following composition:

$$A \times A \stackrel{r \times 1}{\to} A^A \times A \stackrel{ev}{\to} A$$

where ev is the evaluation map. We formalize this example in Section 6.2.2.

(iii) Domain Theory and Untyped Lambda Calculus. Note that reflexive object can also arise from a different approach to studying computation. For example, to model untyped lambda calculus, we require the isomorphism $A^A \times A \simeq A \times A$. Categories of domains which specify denotational semantics for untyped lambda calculi contain such objects to capture the application of $a : A$ to $b : A$ as is permitted in an untyped lambda calculus.

Again, this application taken together with such an object A, as well as all the retracts of A, gives a Turing category. For more details on domains and denotational semantics [42].

Chapter 3

Ranges in Turing Categories

In this chapter, we will re-examine the material from my M.Sc. thesis research ([47], Chapter 2).

Recall that there are two equivalent definitions for a recursively enumerable set — the domain of a recursive map, or the range of a total recursive map. The very definition of a restriction category allows us to talk about domains of maps; however, the notion of range is not part of the structure. Ranges in restriction categories take the form of restriction idempotents, and are related to a topological concept of open maps. The concept of ranges in a category is, in a sense, dual to that of restriction idempotents. The axioms for a category with ranges are described in [13]. In this chapter, the overlap of Turing and range categories is outlined.

3.1 Range Categories

Consider the range of a mapping $f : A \to B$ in Par. It is the largest subset S of B such that for every $y \in S$ there exists an $x \in A$ such that $f(x) = y$. Now, in line with the concept of domains of maps expressed as restriction idempotents, the range of f can also be viewed as an idempotent $\hat{f}$, with

$$\hat{f}(y) = \begin{cases} y & \text{if } \exists x \in B \text{ such that } f(x) = y \\ \uparrow & \text{otherwise.} \end{cases}$$

To explore ranges in categories other than Par, maps must again have corresponding range idempotents, described in the following definition.

Definition 3.1.1 A restriction category C is called a **range category** if it has an operator $\widehat{(-)}$, which sends $f : A \to B$ to a map $\hat{f} : B \to B$, and satisfies the following axioms:

[**RR.1**] $\overline{\hat{f}} = \hat{f}$

[**RR.2**] $\hat{f}f = f$

[**RR.3**] $\widehat{\overline{g}f} = \overline{g}\hat{f}$ with $\mathrm{cod}(f) = \mathrm{dom}(g)$

[**RR.4**] $\widehat{g\hat{f}} = \widehat{gf}$ with $\mathrm{cod}(f) = \mathrm{dom}(g)$

This range theory captures many properties of the conventional concept of range in Par. However, often a fifth axiom is added:

[**RR.5**] $f\hat{g} = h\hat{g}$ whenever $fg = hg$. In Par, this holds true. Suppose $fg = hg$, with $f, g, h \in \mathsf{Par}$, then

$$f\hat{g}(y) = \begin{cases} f(y) & \text{if } \exists x \in B \text{ such that } g(x) = y \\ \uparrow & \text{otherwise.} \end{cases}$$

$$= \begin{cases} h(y) & \text{if } \exists x \in B \text{ such that } g(x) = y \\ \uparrow & \text{otherwise.} \end{cases} = h\hat{g}(x)$$

This implies, by extensional equality, exactly that $f\hat{g} = h\hat{g}$. With this axiom, certain non-intuitive examples of ranges are eliminated, namely when $\overline{f} = 1$ for all $f \in \mathsf{C}$, so that $\hat{f} = 1$. In a category with ranges, any morphism $r : X \to Y$ such that $\hat{r} = 1_Y$, in particular, any retraction, is called a **surjection**. Thus, intuitively, RR.5 says that every morphism is surjective onto its range.

RR.3 and RR.4 are predicates that concern more than one map in a range category. However, it may be the case, in a cartesian restriction category C, that it would be desirable to define some semblance of range for a single map in the category, independently of other maps with which it may compose. Recall from the previous chapter that restriction idempotents are open, including range maps, and that they compose to form open maps. The (always non-empty, containing the identity maps) subcategory of C of open maps is in fact a range category, with $\exists_f(e) = \widehat{fe}$, where e is a restriction idempotent.

With this definition, it follows that ranges of maps are defined by $\hat{f} = \exists_f(1)$. From now on, for an open map f the notation $\widehat{f}, f : A \to B$, will refer to the range of f in the open map subcategory of C, where $\widehat{f}$ is defined. Sometimes, we will say that a category satisfies RR.5 even if it is not a range category. By that, we mean that the subcategory of open maps satisfies RR.5.

Such notation helps combine the original definition of a range category and the openness of certain maps to show a category has a range combinator (as in the upcoming proof). Since the range combinator is in fact unique once a restriction combinator in the category is fixed, this means that there is no ambiguity in writing $\hat{f}$ when f is open.

Before exploring ranges in Turing categories, another important point to note about range maps is their connection to partial inverses. We first recall the definition of a partial isomorphism:

Definition 3.1.2 (Cockett, Guo and Hofstra, [13]) A **partial inverse** of a map $f : X \to Y$ in a cartesian restriction category C is a map $f^{-1} : Y \to X$ such that:

(i) $f^{-1}f = \overline{f}$

(ii) $ff^{-1} = \overline{f^{-1}}$

Then, if f has a partial inverse it is called a **partial isomorphism**.

As with ordinary isomorphisms, a partial inverse to a map is necessarily unique if it exists. Not all maps have a partial inverse, e.g. a non-injective map in Par does not.

The range combinator is, in a sense, the dual concept to the restriction combinator. That is, reversing the arrows in a range category results in a range category where the domain of the opposite arrow of a map f (which is also the same as the domain of the partial inverse of f) is in fact the range of f. The following proposition ties together the concepts of partial inverses and ranges, expressing exactly this property.

Proposition 3.1.3 (Cockett, Guo and Hofstra, [13]) Suppose C is a cartesian restriction category. Then, if a map $f : X \to Y$ in C has a partial inverse, it is open.

We also give the proof because it will be useful for the formalization of the above proposition in the finished project.

Proof: Let f^{-1} be a partial inverse of $f : X \to Y$. For $e \in \mathcal{O}(X), e' \in \mathcal{O}(Y)$, define

$\exists_f(e) = (f^{-1})^*(e)$, and verify the open map axioms:

$$\begin{aligned} \textbf{[O1]}.\ e' &\geq e'\overline{f^{-1}} \\ &= \overline{e'\overline{f^{-1}}} \\ &= (ff^{-1})^*(e'), \text{as} ff^{-1} = \overline{f^{-1}}, \text{so that} \end{aligned}$$

$$\begin{aligned} \exists_f(f^*(e')) &= (f^{-1})^*(f^*(e')) \\ &= (ff^{-1})^*(e') \\ &\leq e' \end{aligned}$$

$$\begin{aligned} \textbf{[O2]}.\ ef^*(e') &= e\overline{e'f} \\ &= e\overline{e'ff} \end{aligned}$$

$$\begin{aligned} &= e\overline{fe'f} \\ &= \overline{f}^*(e)\overline{e'f} \\ &= (f^{-1}f)^*(e)f^*(e') \\ &= f^*(f^{-1})^*(e)f^*(e') \\ &= f^*((f^{-1})^*(e)e') \\ &= f^*(\exists_f(e)e') \end{aligned}$$

$$\begin{aligned} \textbf{[O3]}.\ e'\exists_f(e) &= e'(f^{-1})^*(e) \\ &= e'\overline{ef^{-1}f^{-1}} \\ &= e'\overline{f^{-1}}(f^{-1})^*(e) \\ &= (\overline{f^{-1}})^*(e')(f^{-1})^*(e) \\ &= (ff^{-1})^*(e')(f^{-1})^*(e) \\ &= (f^{-1})^*(f^*(e')e) \\ &= \exists_f(f^*(e')e) \end{aligned}$$

∎

3.2 Ranges and Retractions

Now, to prepare for describing Turing categories with ranges, the following lemma about ranges and idempotents will help verify the necessary technical details.

Lemma 3.2.1 (Vinogradova, [47]) Suppose (m, r) is an embedding-retraction pair of X into $A \in \mathsf{C}$, and the idempotent mr is open. Then

(i) m is open, and $\hat{m} = \widehat{mr}$

(ii) r is open, and for any $e \in \mathcal{O}(A), \exists_r(e) = m^* \exists_{mr}(e)$

(iii) for any $e \in \mathcal{O}(A)$, when $r = r\hat{m}$, $\widehat{re} = rem$

(iv) the map $r' = r\widehat{m}$ is also a retraction for X, and, assuming RR5, mr' is a restriction idempotent

We have formalized this lemma and proved part (iv). This is described in Section 5.6.1. The proof is as follows.

Proof:

(i) First, define $\widehat{me} = \exists_m(e) = \exists_{mr}(r^*(e))$, and check the axioms for $e \in \mathcal{O}(A), e' \in \mathcal{O}(X)$:

$$\begin{aligned} \textbf{[O1]}.\ \exists_m(m^*(e')) &= \exists_{mr}(r^*(m^*(e'))) \\ &= \exists_{mr}((mr)^*(e')) \\ &\leq e' \end{aligned}$$

by O1 for the open map mr.

[O2]. As mr is open, by O2, it follows that

$u \wedge (mr)^*(u') \leq (mr)^*(\exists_{mr}(u) \wedge u')$

Then, taking $u = r^*(e), u' = e'$,

$$\begin{aligned} u \wedge (mr)^*(u') &= r^*(e) \wedge (mr)^*(e') \\ &= r^*(e) \wedge r^* m^*(e') \\ &= r^*(e \wedge r^* m^*(e')), \text{and} \\ (mr)^*(\exists_{mr}(u) \wedge u') &= r^* m^*(\exists_{mr}(r^*(e)) \wedge e') \\ &= r^* m^*(\exists_m(e) \wedge e'), \text{so that} \\ r^*(e \wedge r^* m^*(e')) &\leq r^* m^*(\exists_{mr}(r^*(e)) \wedge e') \end{aligned}$$

Now, r^* preserves order, as for any restriction idempotents e_1, e_2, the following sequence of inequalities holds

$$\begin{aligned}
r^*(e_1) &\leq r^*(e_2) \\
\overline{e_1 r} &\leq \overline{e_2 r} \\
r\overline{e_1 r} &\leq r\overline{e_2 r} \\
e_1 r &\leq e_2 r, \text{which is equivalent to saying} \\
e_2 r\overline{e_1 r} &= e_1 r \\
e_2 e_1 r &= e_1 r, \text{and since } r \text{ is an epimorphism,} \\
e_2 e_1 &= e_1, \text{so that exactly} \\
e_1 &\leq e_2
\end{aligned}$$

We then conclude that $e \wedge r^* m^*(e') \leq m^*(\exists_{mr}(r^*(e)) \wedge e')$, which implies $e \wedge (mr)^*(e') \leq m^*(\exists_m(e) \wedge e')$.

[O3]. By axiom O3 for mr,

$$u' \wedge \exists_{mr}(u) \leq \exists_{mr}((mr)^*(u') \wedge u)$$

Now, let $u = r^*(e'), u' = e$, so that

$$\begin{aligned}
e' \wedge \exists_{mr}(r^*(e)) &\leq \exists_{mr}(r^* m^*(e') \wedge e) \Rightarrow \\
e' \wedge \exists_m(e) &\leq \exists_m(m^*(e') \wedge e)
\end{aligned}$$

as needed.

Finally, $\hat{m} = \exists_{mr}(r^*(1)) = \widehat{mr\overline{r}} = \widehat{mr}$.

(ii) For r, define: $\exists_r(e) = m^* \exists_{mr}(e)$, and check the axioms for $e \in \mathcal{O}(A), e' \in \mathcal{O}(X)$:

[O1]. By axiom O1 for mr, with idempotents $e, r^*(e') \in \mathcal{O}(A)$,

$$\exists_{mr}((mr)^*(r^*(e'))) \leq r^*(e'), \text{ applying } m^*,$$

$$\begin{aligned} m^*\exists_{mr}((mr)^*(r^*(e'))) &\leq m^*r^*(e'), \text{ so that} \\ \exists_r((rmr)^*(e')) &\leq (rm)^*(e') \text{ as } rm = 1, \\ \exists_r(r^*(e')) &\leq e' \end{aligned}$$

[**O2**]. By O2 for mr, with idempotents $e, r^*(e') \in \mathcal{O}(A)$, we get

$$\begin{aligned} e(mr)^*(r^*(e')) &\leq (mr)^*(\exists_{mr}(e)r^*(e')) \Rightarrow \\ er^*(e') &\leq r^*(m^*\exists_{mr}(e)m^*r^*(e')) \\ e \wedge r^*(e') &\leq r^*(\exists_r(e) \wedge e') \end{aligned}$$

[**O3**]. Once more, by O3 for mr, with idempotents $e, r^*(e') \in \mathcal{O}(A)$, conclude

$$\begin{aligned} r^*(e')\exists_{mr}(e) &\leq \exists_{mr}((mr)^*r^*(e')e), \text{ applying } m^* \\ m^*(r^*(e')\exists_{mr}(e)) &\leq m^*\exists_{mr}(r^*(e')e) \\ m^*r^*(e')m^*\exists_{mr}(e) &\leq m^*\exists_{mr}(r^*(e')e) \text{ so that exactly} \\ e' \wedge \exists_r(e) &\leq \exists_r(r^*(e') \wedge e) \end{aligned}$$

(iii) For any $e \in \mathcal{O}(A)$,

$$\overline{re}m = rm\overline{re}m = r\overline{re}m = r\bar{r}em = rem$$

then, the following sequence of equalities holds, proving the result:

$$\widehat{re} = \widehat{re\widehat{m}} = \widehat{rem} = \widehat{\overline{rem}} = \overline{rem} = rem$$

(iv) (m, r') is indeed an embedding-retraction pair of X into A:

$$r\widehat{m}m = rm = 1_X$$

Now, to show that $mr\widehat{m} = mr\widehat{mr}$ is indeed a restriction idempotent,

$$mr = mrmr = 1_Amr$$

and applying RR5:

$$mr\widehat{mr} = \widehat{mr}$$

For an object X with an open embedding-retraction pair (m, r), it is safe to consider the retraction $r = r\widehat{m}$, as such a morphism always exists. When RR5 holds, one may assume $mr = \overline{mr}$, as such an idempotent exists.

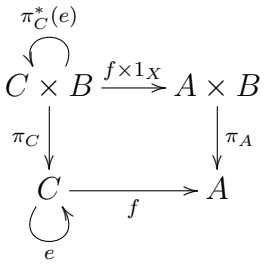
3.3 The Beck-Chevalley Condition

The Bech-Chevalley condition has long been studied in a variety of contexts, and the following definition, found in [13], introduces it in the context of ranges and restriction products. This condition will be useful in describing Turing categories with ranges.

Definition 3.3.1 A cartesian restriction category C is a **cartesian range category** when it is a range category and the following condition, known as the **Beck-Chevalley condition** [13]:

$$\exists_{f \times 1_X}(\pi_C^*(e)) = \pi_A^*(\exists_f(e))$$

holds true for all (pullback) squares of the form



An arbitrary cartesian restriction category C is said to satisfy the BCC whenever Open(C) is a cartesian range category.

To say C satisfies BCC is equivalent to saying that for all open maps $f, g \in \mathsf{C}, \widehat{f \times g} = \hat{f} \times \hat{g}$. Indeed, suppose $\widehat{f \times g} = \hat{f} \times \hat{g}$. Then

$$\begin{aligned} \exists_{f \times 1_X}(\pi_C^*(e)) &= \exists_{f \times 1_X}(e \times 1_X) \\ &= (\widehat{f \times 1_X})(\widehat{e \times 1_X}) \\ &= \widehat{fe} \times 1_X (\text{by assumption}) \\ &= \pi_A^*(\exists_f(e)) \end{aligned}$$

And conversely, when $\exists_{f \times 1_X}(\pi_C^*(e)) = \pi_A^*(\exists_f(e))$,

$$\begin{aligned} \widehat{(f \times 1)} &= \hat{f} \times 1, \text{and} \\ \widehat{(1 \times g)} &= 1 \times \hat{g}, \text{so} \\ \widehat{f \times g} &= \widehat{(f \times 1)(1 \times g)} \\ &= \widehat{(\widehat{f \times 1})(\widehat{1 \times g})} \\ &= \widehat{(\hat{f} \times 1)(1 \times \hat{g})} \\ &= \widehat{\hat{f} \times \hat{g}} \\ &= \hat{f} \times \hat{g} \end{aligned}$$

Generalizing to arbitrary finite products, it follows that for any $f =< f_i, ..., f_n >: C \to \Pi_{i=1}^n A_i$ in $\mathsf{C}, \hat{f} =< \hat{f}_i, ..., \hat{f}_n >$. The following statement is true about range categories and their subcategories:

Lemma 3.3.2 Consider a range category C with a subcategory D. Then D satisfies the Beck-Chevalley condition if C satisfies the Beck-Chevalley condition.

The proof is immediate, since the inclusion $\mathsf{D} \hookrightarrow \mathsf{C}$ preserves the ranges and the products. This lemma is applicable to the case of a Turing cartesian range category with PCA $\mathbb{A}$, and its subcategory Comp($\mathbb{A}$).

3.4 Ranges and PCA's

We now investigate conditions on a PCA under which the corresponding Turing category will have ranges. A formalization of this proposition is found in Section 5.6.2.

Proposition 3.4.1 (Vinogradova, [47]) Suppose C is a Turing category with a universal application $\bullet : A \times A \to A$ which is open. Suppose also that every code (total map) $c_f : 1 \to A$ is open, and assume BCC. Then the categories Comp($\mathbb{A}$),C, and Split(Comp($\mathbb{A}$)) are range categories, and

$$\mathrm{Comp}(\mathbb{A}) \hookrightarrow \mathsf{C} \hookrightarrow \mathrm{Split}(\mathrm{Comp}(\mathbb{A}))$$

are range preserving inclusions.

Proof: First, note that the assumptions in the proposition imply that every computable map $f : A \to A$ is open, as it can be expressed as $\bullet\langle c_f!_A, 1_A\rangle$, which is a composition of an open map and a product of open maps (open by BCC). That means in particular that every computable idempotent on A is open. By the previous lemma it follows that all embedding-retraction pairs are open as well.

An arbitrary map $f : X \to Y$ is again a composition of open maps, with $f = r_Y \bullet \langle c_f!_A, 1_A\rangle m_X$.

∎

To avoid ambiguity in notation, the range of a composition of several maps will be denoted by $(-)_{ran}$ in the upcoming sequence of equalities. Now, for $f : X \to Y$ in C, the range can be computed as follows:

$$\hat{f} = (r_Y \bullet \langle c_f!_A, 1_A\rangle m_X)_{ran}$$

Note that $f = r_Y\widehat{m_Y}m_Y r_Y \bullet \langle c_f!_A, 1_A\rangle m_X$, so that $c'_f \cdot - = m_Y r_Y(c_f \cdot -)$ is another code for f. Then, without loss of generality, we can assume $r = r\hat{m}$.

By the previous lemma, with $e = \widehat{(c_f \cdot -)\widehat{m_X}} = \widehat{(c_f \cdot -)m_X}$, which is the range of a composition of two open maps, the result follows:

$$\begin{aligned} \hat{f} &= (r_Y \widehat{(c_f \cdot -)m_X})_{ran} \\ &= r_Y((c_f \cdot -)m_X)_{ran} m_Y \\ &= r_Y((c_f \cdot -)\widehat{m_X r_X})_{ran} m_Y \end{aligned}$$

Next, we consider what happens in the underlying partial combinatory algebra side of things in terms of combinators.

Definition 3.4.2 Suppose $\mathbb{A} = (A, \bullet)$ is a PCA in a cartesian restriction category C. Then

(i) $\mathbb{A}$ has (**weak**) **range combinators** whenever for every code $a : 1 \to A$ there exists a combinator r_a such that $(r_a \cdot a) \cdot - = \widehat{a \cdot -}$.

(ii) $\mathbb{A}$ has a **strong range combinator** if there exists a combinator r such that $(r \cdot a) \cdot - = \widehat{a \cdot -}$ for every code $a : 1 \to A$.

Using the above results, the next theorem gives a characterization of when a Turing category is a range category.

Theorem 3.4.3 (Vinogradova, [47]) Consider a cartesian restriction category C with a PCA $(A, \bullet)$. Then A has weak range combinators if and only if $\mathrm{Comp}(\mathbb{A})$ is a range category. Furthermore, $\mathrm{Split}(\mathrm{Comp}(\mathbb{A}))$ is then also a range category.

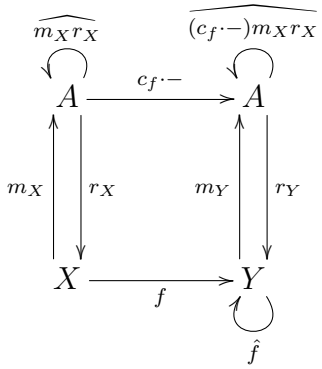
Proof: Suppose $\mathrm{Comp}(\mathbb{A})$ is a range category, with a PCA $(A, \bullet)$, which contains a code $a : 1 \to A$. Then the map $a \cdot - : A \to A$ is open, so let b be a code for $\widehat{a \cdot -}$. Let r_a be the combinator for the lambda calculus expression $\lambda x.b$, so that $r_a \cdot a = b$. It follows that $\widehat{a \cdot -} = b \cdot - = (r_a \cdot a) \cdot -$, so that $\mathrm{Comp}(\mathbb{A})$ has a weak range combinator.

It follows from the definition that having weak range combinators in $\mathbb{A}$ means that for any map $f : A \to A$ in $\mathrm{Comp}(\mathbb{A})$, with code a, there exists a range combinator r_a, with $\widehat{f} = (r_a \cdot a) \cdot -$, so that it is open.

Then, as shown in the previous proposition, every map $X \to Y$, where X, Y are retracts of A, must also be open. It follows that Split(Comp($\mathbb{A}$)), as well as Comp($\mathbb{A}$) are range categories.

∎

Having a strong range combinator in $\mathbb{A}$ allows us to compute the range of a map in a Turing category in a uniform manner. Thus, for any $f : X \to Y$, the range of f may be expressed as $\hat{f} = r_Y \widehat{f m_X r_X} m_Y$, illustrated as follows,



Let c_{mr} be a code for $m_X r_X$, and c_f be a code for f. At a point (map) x,

$$\begin{aligned} c_f(c_{mr}x) &= ((kc_f)x)(c_{mr}x) \\ &= s(kc_f)c_{mr}x, \text{ now applying the range combinator,} \\ \widehat{fm_X r_X} &= r(s(kc_f)c_{mr})\cdot -, \text{ so that} \\ \hat{f} &= r_Y(rs(kc_f)c_{mr}\cdot -)m_Y \end{aligned}$$

Example 3.4.4 The standard model Comp($\mathbb{N}$) is a range category, with range combinator inherited from Par. The underlying PCA in fact has a strong range combinator r, which may be expressed using pseudo-code as follows:

IsInRange (n, x) {

For(int $i = 0$ to ∞) {

For(int $j = 0$ to i) {
Do 1 step of each computation $\phi_n(j)$
If $\phi_n(j)$ halts at this step {
Test $\phi_n(j) = x$
If TRUE, return x
}
}
}
}

The formalization of range structure in Comp($\mathbb{N}$) is discussed in Section 6.2.4. This algorithm computes the value of the map $n\cdot - : \mathbb{N} \to \mathbb{N}$ at each integer i sequentially, using interleaving, and tests the equality $n\cdot i = x$. The algorithm halts if the equality holds for some i, and outputs x. It continues testing infinitely if x is not in the range of n.

For each n, this computation corresponds to some (computable) function of x, with code $\phi_r(n)$ by the S_n^m theorem [20]. This amounts to saying exactly that $(r\cdot n)\cdot x = \widehat{\phi}_n(x)$.

Note that in the computation, a test for equality, $\phi_n(i) = x$, takes place. This test is indeed possible in $\mathbb{N}$, but such equality tests may not exist in other PCA's, which would make a similar computation strategy infeasible.

Chapter 4

Formalization and Coq

4.1 Formalization

Traditionally, proofs and definitions in mathematics are recorded in a format such that their meaning is essentially unambiguous to a human reader, even though certain aspects, such as isomorphisms or the semantics of the equality comparison, may be suppressed or assumed. Allowing this to be the case is the forgiving nature of natural language, and is not acceptable in a formal setting. While most of the theorems established in this way are most likely true, with their relatively informal proofs containing no errors, there are benefits to taking a more error-proof and mechanized approach to many areas of mathematical inquiry. Formalization of new and existing theories aims to do just that.

A formal approach to theorem proving involves expressing the desired aspects of a certain theory by means of a formal language. The contents of a theory (i.e. definitions, propositions and proofs thereof) must be constructed as well-formed formulas, made up of the symbols of the selected formal language and in accordance with its rules. Formalization is an extremely reliable and rigorous approach to studying mathematical constructs. In contrast with informal proofs of mathematical results, a

formal one is guaranteed to be correct so long as the verification strategy is correct.

The other key advantage of formalization over informal proofs is the possibility of automating certain parts of a proof that may take a prohibitively long time to complete by hand. For this reason, the result that has largely contributed to putting Coq (as well as the very idea of formalization) on the map as a way to approach doing mathematics is the proof of the Four Colour Theorem, formalized in Coq in 2008 by Georges Gonthier [24]. The theorem states that four colours are sufficient to colour all the vertices of a planar graph such that no two adjacent ones are of the same colour. The previously incomplete part of the proof of this theorem hinges on checking each of the thousands of cases that arise as possible configurations in the colouring of the graph.

In order to complete the proof, the researchers have constructed a formal language description of the problem and of each of these cases. They then devised an automated verification strategy that was able to handle all the cases. Since there was no way to check this proof by hand (due to the sheer volume of computation required), it was not immediately accepted as correct by the entire mathematics community. However, it was a milestone achievement for machine-assisted theorem proving, demonstrating its applicability to an unsolved problem of current interest.

The choice of logic for the formalization and the particular software that is built based on it, as well as the choice to use an existing library of formalized category theory has a huge impact on the type of challenges that arise in the process of formalizing further concepts in recursion and category theory. The following section outlines certain features of the chosen software and libraries that guided the formalization process.

4.2 Logical Framework behind Coq

The formal language of which Coq is an implementation, the calculus of (co-)inductive constructions, was originally introduced in the paper "The Calculus of Constructions" by Thierry Coquand and Gerard Huet, published in 1988 [19]. This language gives a constructive foundation for mathematics based on intuitionistic logic, meaning that the law of excluded middle does not hold, and the only way to prove the existence of a term with certain properties is to explicitly define such a term.

Now, cartesian closed categories constitute a model for computation in the simply-typed lambda calculus (STLC) [34]. This serves as further motivation to consider Turing categories from the point of view of formal language description via CIC due to the parallels between CCC and Turing categories.

The Barendregt lambda cube is a formulation of a variety of lambda calculi of varying expressive power, capable of expressing the following forms of expressing following forms of abstraction in addition to the standard lambda calculus structure [2]:

(i) Polymorphism: terms depending on types

(ii) Dependent types: types depending on terms

(iii) Type operators: types depending on types

CIC is located at the top of Barendregt's lambda cube, meaning that it is able to express all three of the above forms of abstraction. Moreover, the CIC contains inductive types (for details, see Section 4.2), making it a very expressive language to use for formalization.

4.2.1 The Barendregt Lambda Cube

The lambda calculus is a formal language based on type theory and capable of expressing a variety of mathematical, logical, and computational concepts through the

use of type formation rules [2].

Computation in simply typed lambda calculus (STLC) takes the form of sequences of rewrites (reductions). There are three rewriting rules in the STLC system, namely α-reduction (renaming of bound variables), β-reduction (which states that $(\lambda x.fx)y$ rewrites to fy), and η-reduction (which states that $\lambda x.fx$ rewrites to f, where $x \notin \mathrm{FV}(f)$). The STLC is a strongly normalizing formal system, meaning that any sequence of rewrites terminates in a normal form.

Now, the expressive power of a simply lambda calculus can be greatly augmented by introducing additional type and term formation rules. This allows us to express the full features of higher order logic. The terms formed via these rules may not be typable in STLC, so we get a more powerful system. The following type formation strategies, mentioned in the introduction, can be added to the STLC rules to build new formal languages [2]:

(i) Polymorphism: terms indexed by types

Adding polymorphic typing rules describing how to build new types out of existing ones is the standard approach to introducing second-order logic into STLC by quantifying over types in terms. So, instantiating all second order quantifiers (over types) with specific (first-order, without quantifiers over types) types within a polymorphic term produces a term that can be explicitly typed in STLC.

An example of a polymorphic type is the (polymorphic) identity function, $(\Lambda A.\lambda x : A.x) : (\Pi A.A \to A)$, which can be used to form the identity function on any type. According to notation convention, Λ is used to quantify over types within a term, and Π denotes quantification over types in the type of such a term. The addition of these formation rules to STLC transforms it into a formal language known as System F (for a detailed description [2]).

(ii) Dependent types: types indexed by terms

This is another strategy for building new types out of existing ones as follows:

If A is a type, and U is a collection of types, we may define $B : A \to U$ such that there $B(x)$ is a type for each $x : A$, formally $\Pi x : A.B(x)$.

Predicates (expressions of the type $A \to Prop$, for some type A) are examples of the kind of structures that can now be formed with the addition of these rules. For example, consider $is\_even : \mathbb{N} \to Prop$. Then, $is\_even(x) : Prop$ can be defined to be the proposition for which a proof $pf : is\_even(x)$ exists whenever $x : \mathbb{N}$ is even.

(iii) Type operators: types depending on types

A concept immediately relevant to the subject of this work, a formal definition of a ho*m*-set of maps from A to B in the category of Set, is an example of this kind of type construction.

$$\Lambda A.\Lambda B.A \to B$$

Note that one way to highlight the difference between type operators and second order polymorphism is to notice that the type $\Pi A.A \to A$ is inhabited by terms that are indexed by types, such as the polymorphic identity function, whereas Λ A. Λ B. $A \to B$ is not yet a type until it is applied to types A and B, corresponding to the type indexed by A and B in the relevant family of types.

The three additions to the typing rules of STLC listed above form the three directions of the axes of the so-called "(Barendregt) Lambda Cube" (for details [2]). The higher order polymorphic dependently typed formal system located in the farthest corner from STLC is in fact the Calculus of Constructions, which means it incorporates features (i), (ii), and (iii) discussed above. Addition of a combination of any two of them to STLC yields an extension which is a distinct and a less expressive system than the Calculus of Constructions (CoC).

4.2.2 Sorts, Universes and Impredicativity

With all the above term and type formation rules within the formal language, types and terms are now all mixed together, and claiming that a type simply has type Type may lead to the creation of so-called "impredicative" definitions, a potentially problematic property described below. In order to control the impredicativity situation, a typing hierarchy exists in the COC (and CIC, the Calculus of Constructions with induction). That is, there exists a set of rules in the language for the stratification of all definitions. The set of these strata is the following:

$S = \{\ Set, Prop, Type(i), \text{for } i \in \mathbb{N}\}$

Here, Set is the type of all small sets — that is, it does not include the set of all sets, which contains itself. Prop is the type of all logical propositions. The types Type(i) are referred to as universes for any $i \in \mathbb{N}$. The set S itself is stratified in the following way:

(i) $Set, Prop : Type(1)$

(ii) $Type(i) : Type(i+1)$

Informally speaking, an impredicative definition is a definition that makes use of a type which is as high or higher in the type hierarchy than the type that is being defined. Such a definition invokes self-reference by instantiating one or more of the quantifiers within its predicate with the object being defined. For example, the set which contains all sets that do not contain themselves is an impredicative definition. Furthermore, it creates a paradox wherein such a set can neither contain itself nor exclude itself.

In order to control paradoxes of this nature, universes exist in the formal language. Specifying over which universes the quantifier can be instantiated can prevent definitions of a type $t : Type(i)$ from containing anything of type $Type(i)$ or

higher, thus limiting impredicativity in the language. Some impredicativity, however, is allowed in the CIC. While Type and Set are both predicative, Prop is, in fact, impredicative — no dangerous paradoxes are formed by allowing a quantifier to be instantiated with a proposition that itself contains a quantifier over all propositions. This is because it is impossible to construct an inhabitant of the impredicative proposition (a proof of it).

Going back on the example from the previous section, we may now show full type information:

$\Lambda A : Type(i).\Lambda B : Type(i).A \to B : Type(i+1)$

The following section elaborates on the treatment of this type hierarchy within Coq.

4.3 The Coq Proof Assistant

The Coq proof assistant implements the Calculus of (co)-Inductive Constructions as the underlying language of type-checking. In addition to the basic rules of the language, however, it comes equipped with a large collection of built-in libraries that formalize a variety of concepts commonly encountered in logic, math and computer science (e.g. natural numbers, booleans, propositional logic, etc.), as well as tools that allow the user to aggregate formalisms into more complex structures or define new types of relationships between them.

The following are features of Coq's libraries that need to be outlined in order to proceed with the exposition of the Turing category formalization project in the subsequent chapters. Particular syntax structure will be explained as needed in later chapters focusing on explaining the Turing category formalization files. Note that from now on we use `this font` to denote Coq code.

Sigma Types. Sigma types are dependently typed pairs where the first term is type `A`, and the second is a predicate `P : A → Prop`. In Coq, we write `sig A P`, or

more suggestively {x:A | P x}, as sigma types are used to define formal 'subsets' of terms of a given type. A term x_pf : sig A P is itself a pair, where the first term is x : A, and the second is a proof of the proposition P x. The term proj1_sig x_pf : A for a sigma-type x_pf : sig A P returns the first term of the dependent pair, and the term proj2_sig x_pf is the proof of the predicate P applied to the first term of the sigma type. The sigma type construct is particularly useful to us in formalizing subcategories, such as the total subcategory, see Definition 2.1.4.

Universe Polymorphism. In addition to the Set, Prop, and Type hierarchy, the following are three examples that illustrate the consistency of the hierarchical relationships between the universes and to specific data types:

(i) nat, bool: Type(0)

(ii) list: Type(0) → Type(0)

In Coq, all definitions are well-stratified according to the hierarchy indicated by the enumeration of the types. This prevents generic constructions on universes that could work at different levels of the stratification. Universe polymorphism is introduced in Coq 8.5 in order to support generic definitions over universes, reusable at different levels. This provides the same kind of code reuse facilities as ML-style parametric polymorphism.

Equality and Formalization in Coq. The intended meaning of equality in informal mathematics and logic is generally the Leibnitz definition of equality. That is, two terms x and y are equal if for all properties P, P is a property of x if and only if P is a property of y. Most often, this definition is sufficient, and is assumed rather than specified. This is a prominent example of the distinction between the levels of rigor in the formal and informal approaches to studying mathematics, and in particular, abstract computability. The strategy for making equality judgments in a formal setting is a major design decision which affects the structure of proofs and definitions.

Now, in Coq, like everything else that is definable, the notion of equality is a formalism of a specific type. The type of equality is

```
forall t:  Type, t → t → Prop
```

We may also have `t:  Set` or `t:  Prop` here. The equality comparison is defined to be reflexive, with the constant `eq_refl` representing the proof of `x = x` when applied to a term `x` of any type. Because of the way inductive types work in the CIC, reflexivity is sufficient as the (unique) defining property of the equality relation in order to prove that it is, in fact, equivalent to the Leibnitz equality formulation. In particular, symmetry and transitivity of equality may be proved directly from reflexivity.

In order to construct a proof of a proposition `a = b` for some `a, b:  A`, the user would normally manipulate the goal using Coq's tactics into a form where reflexivity can be employed, then use `eq_refl` to complete the proof. The notion of equality, defined in this way and denoted "=", is well-suited for use in a general categorical setting. For example, stating that map composition in a category is associative can be done using "=". For working with specific categorical examples in a formal setting, however, additional axiomatized propositions may be required to reflect the desired properties of what is simply "=" in the informal sense. For example, equality in the category Set is extensional. So, in order to compare two maps between sets, functional extensionality may be added:

```
AXIOM functional_extensionality:  forall (X Y:Set),
forall (f g:X → Y), (forall (x:  X), f x = g x) → f = g
```

This axiom is saying that two maps $f, g : X \to Y$ are equal whenever they evaluate to the same value at every element of set X.

As part of our formalization, we are building a category where we are reasoning about the equality of partial maps, and, as a consequence, about proofs of the membership of a given element in the domains of the maps we are comparing. To make this easier, we may want to simply identify all proofs of a particular proposition as

equal. This is known as proof irrelevance:

```
AXIOM pf_ir:  forall (A: Prop) , forall (p q:  A), p = q.
```

These axioms are in fact pre-defined in Coq [18] and may be inserted into a code file for use at any time as they do not compromise the consistency of the system.

For definition of more complicated categorical examples than sets and (total) maps, there are other options for handling equality comparisons. One option is to construct an entirely different definition of (setoid) equivalence and use it in place of "=" to compare categorical maps. A setoid is a triple consisting of a type, a relation on the type and a proof that the relation is an equivalence. When a setoid is defined, its equivalence relation can be used to preform rewriting in certain cases, similar to how '='-rewriting is done. In addition, it may be required to satisfy further congruence properties necessary for the kind of rewriting that is informally allowed in a specific categorical situation. For example, map equivalence is usually required to be a congruence relation with respect to composition, i.e., for maps $f, g : A \to B$, $h : C \to A$, $j : B \to D$

$f = g \Rightarrow fh = gh$ and $jf = jg$

Additional congruence relations may be required as further categorical structure is introduced into a category. For example, with the presence of a restriction combinator in a category, we must have that for any maps $f, g : A \to B$, $f = g \Rightarrow \overline{f} = \overline{g}$. So, when setoid equivalence is used, a formal proof of this congruence would be required when instantiating a restriction category. This setoid equivalence approach is employed in some existing category theory libraries (see Section 4.4), but not in the library we have selected. There does, however, exist a formalization of a category of setoids that reflects the nature of setoids relations [39].

The reason we have not made use of the different types of relations modeled within such a category is that we do not require all of the categorical machinery developed in this work. There would be no benefit to this additional layer of structure in the abstract categorical definitions and proofs.

Another option is to axiomatize the desired notion of equality directly in the code file (similar to the two axioms presented above). This is the route we have selected for this thesis. A detailed discussion of the design decisions regarding equality within our formalization will be outlined in the description of the selected category theory library in Section 4.4 as well as in the categorical example formalization overview in Section 6.1.

Recall, also, that informally, a range category may contain an equality restriction idempotent $\hat{\Delta}$ [47]. However, formally defining the meaning of equality within a category is distinct from demonstrating the existence of the equality map in the category.

Type Classes. Type classes are a fairly new feature in Coq. The purpose of type class systems is to allow a form of generic programming for a *class* of types. Classes are presented as an interface comprised of terms specific to a type as well as proofs about those functions as well. One can write programs that are polymorphic over any type which has an instance of the class, and later on, instantiate those to a specific type and implementation (called instance) of the class [6]. They are best explained with an example from this library to illustrate the concept. Here is the formal definition of a category using type classes. Note that the notation '(*...*)' in Coq denotes comments for the reader, and the contents between '(*' and '*)' is not part of the code.

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

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

     (* composition of morphisms:  *)
compose:  ` a b c:  Obj, Hom a b → Hom b c → Hom a c where "f o g":= (compose g f);
```

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

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

     (* identity morphisms:  *)
id:  ` a:  Obj, Hom a a;

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


     (* id right unit:  *)

id_unit_right:  ` (a b:  Obj) (h:  Hom a b), h o id = h

}.
```

To further clarify notation, the symbol ` in front of an argument (such as `a: Obj` in `id`) denotes that this argument is implicit. That is, it does not have to be given explicitly when instantiating the term. This is possible because the Coq system is able to infer the types of these implicit terms (based on the explicit arguments given). This special notation for implicit arguments provides additional flexibility in writing types in Coq. Consider the type of compose. The usual way of presenting its (dependent) type without the use of implicit arguments would be:

```
compose :  forall a:Obj, forall b:Obj, forall c:Obj, Hom a b → Hom b c → Hom a c.
```

This is a function that takes five arguments. Coq allows the flexibility of writing this also as:

```
compose:  (a:Obj) (b:Obj) (c:Obj) :  Hom a b → Hom b c → Hom a c.
```

where the first three arguments appear to the left of a colon, similar to how arguments (formal parameters) to a program in a functional programming language are written.

When arguments are all of the same type, as above, we can write `(a b c: Obj)` as an abbreviation, and it is also possible to remove the parentheses around these arguments and write

```
compose: a b c: Obj : Hom a b → Hom b c → Hom a c
```

Then simply by adding ' to the front, we tell Coq that these arguments are implicit, so instead of writing `(compose a b c g f)`, we must write `(compose g f)`, and from the types of `f` and `g`, Coq can infer `a, b`, and `c`. However, when these implicit arguments are listed in curly braces, such as `'{a: Obj}`, the user has the option of giving the argument explicitly. Coq also allows users to define their own notation, such as using the symbol ∘ for `compose`, as seen in the above definition of category using type classes.

Note that the declared class `Category` is of type `Type`, hence the name 'type classes'. This basic structure of a type class, then, is the declaration and all the terms contained inside. When instantiating the class, the terms whose types are not propositions (in this case, objects, morphisms, identity and composition) must be given, for the corresponding types. Then, proofs of associativity, composition with identity, etc. must be completed in order to supply terms of the type of the corresponding propositions.

Type Class Coercions. Coercions can be defined when it is convenient to implicitly treat an instance of a type class as a term of another type, such as one of the (non-Prop type) terms within the class, or as an instance of another class. For example, when defining a type class to reflect the structure of the (category-theoretic) product, in most cases we want to treat the product of two objects A and B of a category C as another object of C. This is because in order to pass it as an argument to any term that expects an object, such as the identity map in a category, `id: forall A: C, A → A`, it must have the type object.

Coercions also enable Coq to consider an instance of a class as an instance of another class or one of the terms in the given class. In the case of category theory,

this reflects the idea that a more specific type of category (one with more structure) can be considered as the more general type of category (e.g. a cartesian category is a category; or a subcategory of a category is again a category). Defining coercions is the formal representation of this informal concept, and significantly improves the usability and legibility of the code.

In order to define a coercion, consider a function that takes an instance of a type class (representing a categorical concept, for the purpose of this work) as an argument. The type of this function must be same type (or type class) as which the argument will be treated. Defining such a function can be implicit if the coercion calls for the given type class to behave as one of its terms as in the example below. Next, it is necessary to state the nature of the coercion explicitly. We define a specific example of a coercion in Section 5.2 once we have establish sufficient notation.

4.4 The Category Theory Library

There are currently not many category theory library options available for Coq. Before starting the formalization, upon doing some research, we have identified two existing category theory libraries which were the most complete. These were:

- [i] The Coq ConCaT contribution, Constructive Category Theory, by Amokrane Saibi [29]
- [ii] "Category Theory in Coq", due to Adam Megacz
  (source code at: http://www.megacz.com/berkeley/coq-categories/)

The former was the preferred choice as it made use of type classes because they are very well suited for formalizing category theory. The reason for this is the way that type classes structure data — similar to how categories and category-theoretic structures are made up of specific types of data satisfying certain equations or constraints (captured by proof obligations). In addition, type classes are preferred over

Coq Structures due to their user-friendly syntax and a number of features desirable for proving results about categories, such as contexts and class hierarchies [18].

As the project developed, it became apparent that the selected library does not contain all the categorical concepts and results needed for the formalization of Turing categories and related structures. In particular, it lacked important definitions such as cartesian restriction categories, etc. Around this time, a new mathematical library was being developed for a heavily modified version of the Coq proof assistant, HoTTCoq, available at

"https://github.com/HoTT/HoTT"

The idea behind this new library, the Homotopy Type Theory (HoTT) library, is to view equality from a topological perspective, as a kind of path homotopy, eliminating distinctions between pairs of objects that were merely isomorphic in the usual sense (see the Univalent Foundations text [44]). This library is currently being developed at, among other places, the Institute for Advanced Studies (IAS) in Princeton. The development of the library as well as its formalization was the initiative of Steve Awodey and Vladimir Voevodsky and their team at IAS. It has become an international undertaking after the publishing of the Univalent Foundations text in 2013 (as well as multiple international conference presentations).

Since giving the right definition of equality did indeed present itself as a non-trivial issue in the process of formalizing categories, this novel approach appeared to be a good option to consider as the library of choice for Turing category development.

A detailed explanation of HoTT and the library based on it is outside the scope of this thesis as the library was not the one we chose to use in the final product formalization. The reason for this is that using the Coq library for this groundbreaking, elegant and innovative way to handle equality comparison (and the new way to do mathematics developed from this idea), effectively, was not necessary for the formalization project undertaken in this thesis. The research goals of this thesis did not require a complete reformulation of existing mathematics, and the current state of

the HoTT Coq library did not provide a user-friendly development experience due to the lack of documentation and complexity of the library in its state at the time. The following is an example of part of the formalization of the category of sets and partial maps in HoTT.

Note that `hProp` is the type used instead of Prop in the HoTT library, and "proofs" of terms of this type are constructed according to the HoTT rules, with equality between them established as path homotopy. Similarly, `hSet` plays the role of `Set` in HoTT. We omit the explanation of the details of this definition, it is provided here for comparison (for the reader familiar with Coq and the nature of HoTT).

```
Local Notation hom A B:= { P: A → hProp & forall x:  A, P x → B }.

Let compose (A B C: hSet) (m0:  hom B C) (m1:  hom A B): hom A C.
exists (fun a => hp { pf:  m1.1 a & m0.1 (m1.2 a pf)} _).
intros a pf.
exact (m0.2 (m1.2 a pf.1) pf.2).
Defined.

Definition par_cat:  PreCategory.

refine (@Build_PreCategory

     (* The type of objects *)

hSet

     (* The type of morphisms *)

(fun A B ⇒  P: A → hProp & forall x:  A, P x → B )

     (* the identity morphism *)

(fun x ⇒ (fun _ ⇒ hp Unit _;

fun x _ ⇒ x))

     (* composition *)

compose

     _ (* associativity *)

     _ (* left id *)

     _ (* right id *)
```

```
- (* morphisms are truncated; gets picked up automatically by typeclass resolution *)).
```

The code above is meant to be an example giving a rough sketch of how to use the HoTT library to formalize Par, so the proofs about associativity and composition with identity have been left out.

A new library formalizing category theory came out for the standard Coq development stream (for version Coq 8.5beta and later) just as working with the HoTT library began to appear impractical after several months of formalization attempts. This library, originally presented in the paper [45], due to Bart Jacobs and Amin Timany, at the The Coq workshop (2015), presents the following benefits as compared to defining a new library or using one of the existing ones listed above:

(i) the structures and design decisions in the selected library all appeared to be a more than satisfactory choice for representing categorical structure

(ii) using an existing library with additional well-known categorical structure formalized will allow us to eventually expand the Turing category and related structure formalization to additional concepts

(iii) using an existing library that is gaining popularity will allow this work to be easily integrated into a more general effort of formalization of category theory and the various aspects of mathematics modeled by it

4.4.1 Library Structure

The structure of the coding project about which this thesis is written is a collection of Coq code files organized in folders according to the nature of category-theoretic concepts they formalize. The code therein makes use of the categorical definitions, lemmas and theorems formalized in the original library code. We have divided the formalization we have built on to of the code library into the following files:

CompA CompN_Cat Range Par_Cat PCA Restriction Turing

Each of these files will be discussed in the subsequent chapters. For details on how to compile these files, see the ReadMe file provided on 'Turing Category Formalization' GitHub page,

"https://github.com/polinavino/Turing-Category-Formalization"

For the reader looking at the code, the original library organization follows. The retrieval date for the category theory library we are using is 2015-05-26, which corresponds to the code found at:

"https://github.com/amintimany/Categories/tree/
5485b5189c2fd38d5725f2685100fd45a0a31c7c"

We did not upgrade to newer versions of the library in the process of working on our formalization because of compatibility issues. Furthermore, we did not require much of the formal theory added to the new version of the library. This version was presented at 7th Coq Workshop, with the corresponding proceedings published [45]. The improvements to the previous version included additional proofs and new structures, which we did not require for the categorical definition in this thesis. Below is a list of the Coq code (*.v) files found in the library and the folders containing them. Here, folder titles are highlighted in bold, and subfolders as well as the files they contain are denoted by '—'.

Adjunction: Adj_Cat Adj_Facts Adjunction Duality Main_Adj

Algebras: Algebras CoAlgebras Main_Alg

Archetypal: — **Examples**: — List_Monoid_Cat Discr Monoid_Cat PreOrder_Cat

Basic_Cons: — **Facts**: — Adjuncts — Equalizer_Monic — Init_Prod — Main_BConsFacts

— Term_Prod Terminal PullBack Product Main_BCons LCCC Facts Exponential_Functor Exponential Equalizer CCC

Cat: Cat_Terminal Cat_Products Cat_Iso Cat_Initial Cat_Exponential_Facts Cat_Exponential Cat_CCC Cat

Category: Subcategory Opposite Morph Main_Category Composable_Chain Category

Coq_Cats: — **Type_Cat**: — Type_Cat — Topos — Sum — SubObject_Classifier — PullBack — LCCC — Initial — GenSum — GenProd — Facts — Equalizer — Complete — CCC — Card_Restriction Main_CoqCats Set_Cat Prop_Cat

Demo: Demo

Essentials: Facts_Tactics Notations Types

Ext_Cons: — **Prod_Cat**: — Main_ExtCPCat — Nat_Facts — Operations_ProdCats — Prod_Cat Arrow Comma Main_ExtCons

Functor: — **Representable**: — Hom_Func — Hom_Func_Prop — Main_FuncRep — Representable Main_Func Functor_Properties Functor_Ops Functor_Image Functor_Extendor Functor Const_Func_Functor Const_Func

KanExt: Presentation Pointwise Main_KanExt LocaltoGlobal LocalFacts Local GlobaltoLocal GlobalFacts GlobalDuality Global Facts

Limits: Pointwise Main_Limits Limits GenProd_GenSum GenProd_Eq_Limits Complete_Preorder

NatTrans: Operation NatTrans NatIso Main_NatTr Func_Cat

Topos: Topos SubObject_Classifier

Yoneda: Yoneda

Chapter 5

Formalizing Abstract Computational Structure

In this chapter, examples from the Coq code will be given to illustrate the approaches, challenges, and results of the formalization process. For the complete code, see

"https://github.com/polinavino/Turing-Category-Formalization"

Specifically, this chapter is about the **Restriction.v, Turing.v, PCA.v, CompA.v**, and **Range.v** files in the GitHub repository.

5.1 Existing Formalizations Work

There has not been much work done on the subject of formalizing specifically the Turing category model of abstract computability; however, there does exist an Agda formalization for restriction categories (see below). Agda is another dependently typed programming language with induction, and a syntax similar to Haskell. Unlike Coq, it has no support for tactics, and proofs are written in a functional programming style.

The aforementioned formalization can be found at

"https://github.com/jmchapman/restriction-categories"
and described in the paper by James Chapman, Tarmo Uustalu and Niccolo Veltri, [7]. The scope of that formalization is greater than that necessary for formally describing Turing category structure. We have formalized many of the same results (in Coq), as those done for this Agda formalization. The Agda project includes results found in the original restriction categories paper by Cockett and Lack [16]. The scope of this Agda formalization is the first two chapters of the original paper. These include the completeness result (stated below) and all the necessary preliminary results and definitions.

Theorem 5.1.1 (Cockett and Lack, [16]) Every split restriction category C is isomorphic, as a restriction category, to a partial map category on its total subcategory for the stable system of monics given by the sections of the restriction idempotents of C.

This Agda formalization is part of a larger formalization work the goal of which was to formalize the Delay monad [8], which is useful in modeling non-terminating behaviors. The authors believe, but do not appear to have shown, that this construction constitutes a Turing category. It would be an interesting pursuit to use our formalization of Turing categories to study whether it indeed admits Turing structure.

We have formalized only the restriction category theory (from the first two chapters of [16]) needed to define Turing categories, as well as a number of results from the first four chapters of the original Turing categories paper by Cockett and Hofstra [15], which are stated in Chapter 2 of this work. We have also formalized a number of original results from Chapter 2 of my Master's thesis [47], and new results developed as a part of this work about additional abstract computational structure (Chapter 4) and examples (Chapter 5).

The design decisions made for the above Agda formalization project include the use of dependent records with fields for the data of the structure and fields for the

equations. In order to represent hom-sets, instead of the use of setoid equality, where the laws are given in terms of the equivalence relation of the setoid, propositional equality (the identity type) is used. For the purposes of this Agda formalization project, using propositional equality requires functional extensionality and quotients (for our discussion of extensionality, see Section 4.3).

These design decisions made in the above Agda project regarding treatment of equality and organization of categorical data have both turned out to be central to our Coq formalization project as well, as described in the upcoming sections of this chapter. Furthermore, the Agda development uses universe-polymorphism, which is now an available feature in Coq (8.5) and allows for more generally applicable definitions while maintaining a strict type hierarchy within each one.

While this is an interesting and detailed project, we believe that a formalization of the particular area of mathematics (and computer science) we have selected would benefit more from being done in Coq. This is largely due to the quality of the category theory library we have chosen to use, as well as the formal definition of partial recursive maps, together with the proof of the S_n^m theorem. These are essential building blocks of a categorical description of abstract and traditional computability.

There is also a formalization done in Isabelle/HOL of several systems of axioms describing a logic for reasoning in a partial categorical setting, done by Benzmuller and Scott [3]. The judgments within the systems of axioms studied are done in terms of Kleene equality rather than Leibnitz equality. The reason for this is that the goal of this work is to generalize axioms for a monoid to a partial composition operation in order to obtain a set of rules that axiomatize category theory. No further development of category theory concepts, such as limits, the Yoneda embedding, etc., were completed. The focus of this work was specifically to compare different sets of axioms as the basis for partial composition in a category.

The approach to working with partiality chosen by the authors works well in an extensional (or set-theoretic) setting, and the examples we formalize in this work

(see Chapter 6) are all of an extensional nature. However, the theory of Turing categories is geared toward being able to abstract computation in a way that forgoes extensional judgments in its presentation. Furthermore, the systems of axioms studied rely heavily on the concept of codomain, in addition to the domain. The abstraction of a codomain is a structure not always present in a Turing category. The study of the conditions for such an abstract development is a subject of investigation on its own.

Another category theory formalization (also done in Isabelle/HOL and described in [31]) focuses on the Yoneda embedding, making extensive use of ZFC set theory. This work, while presenting an interesting study of the Yoneda functor, does not include key categorical constructs that we have made use of in our work such as limits, cartesian closedness, etc. The definitions formalized are strictly those needed to demonstrate the Yoneda embedding (i.e. hom-sets, etc.). The category theory formalizations mentioned above are the most relevant to our work; for a broader survey of existing formalizations, see [38]. Overall, the advantage of using Coq over another formal language is due to a combination of its detailed documentation, relative popularity in light of recent proofs of well known problems (such as the four colour theorem), existing formalizations of category theory and partial recursive maps, and the convenience of certain features of type classes for our own formalization.

5.2 Using Type Classes to Formalize Partiality

In this section, we describe the specifics of our formalization of the categorical concepts, definitions and theorems related to Turing categories. The types of categories we formalize here can be arranged into hierarchies, arranged from least additional structure to most, as follows:

$$\text{Restriction Categories} \subseteq \text{Cartesian Restriction Categories}$$

$\subseteq$ Turing Categories $\subseteq$ Turing Range Categories

Restriction Categories $\subseteq$ Range Categories $\subseteq$ Cartesian Range Categories

Following the process of capturing categorical concepts and structure in the category theory library on top of which the abstract computation formalization is built, we first build a type class to represent the main tool for modeling partiality in a category, the restriction combinator (abbreviated 'r.c.'). Its type must capture that a restriction combinator takes a map in a given category, and returns another map from the source object of the original map to itself, which satisfies the four properties in Definition 2.1.2. The restriction combinator is defined to be:

```
Definition rcType:  Category → Type :=

fun (C:Category) ⇒ forall a b:C, Hom a b → Hom a a.
```

A Coq definition has three parts, the name, its type, and a lambda term which is the body of the definition, with : and := as separators. In general, the notation

```
fun (x1:A1) ...  (xn:An) ⇒ ...
```

in Coq denotes a lambda term of the form $\lambda x_1 : A_1$. ... $\lambda x_n : A_n$... . .... As in the definition of `Category` in Section 4.3, Coq allows some additional flexibility in presenting types. The above definition can be written as:

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

Here the argument `C` is introduced on the left of the colon (with its type) and the `fun (C:Category) ⇒` on the right is omitted. Notation for implicit arguments is also allowed in such definitions.

As mentioned, Coq proofs are carried out by starting with the statement of a theorem or lemma as a goal, and applying tactics, which reduce the goal to subgoals, repeating until all subgoals are proved. It is also possible to use tactics to fill in the body of a definition or an instance of a type class. As a simple example, we could

write the above definition as follows:

```
Definition rcType (C: Category) :  Type.

exact (forall a b:C, Hom a b -> Hom a a).
```

Only the name and type are given on the first line. The exact tactic fills in the body directly. In general, we use this capability for more complex definitions that are filled in with a series of steps using tactics. Now, the type class corresponding to the informal concept of a restriction combinator is as follows.

```
Class RestrictionComb `(C: Category):  Type:=

{

     rc:  rcType C ;

     rc1 :  forall (a b:  Obj), forall (f:  Hom a b), f (rc a b f)= f ;

     rc2 :  forall (a b c:  Obj), forall (f:  Hom a b)(g:  Hom a c),

          (rc a c g ) (rc a b f ) = (rc a b f )  (rc a c g ) ;

     rc3 :  forall (a b c:  Obj), forall (f:  Hom a b)(g:  Hom a c),

          rc a c (g (rc a b f) ) = (rc a c g )  (rc a b f ) ;

     rc4 :  forall (a b c:  Obj), forall (f:  Hom a b)(g:  Hom b c),

          (rc b c g) f = f (rc a c (g f)) ;

}.
```

In addition to defining the `RestrictionComb` type class, it is useful to define a coercion in order to be able to treat an instance of this class as a term of type `rcType C` for some `C: Category`. This improves clarity of the code and makes it much easier to read and understand. This (and other coercions) allow us to forgo referring to a specific term in a type class (which is rather notation-heavy), and is an intuitive way to work with type classes. In the case of a restriction combinator, for example, we know that it must satisfy the propositions (that is, proof obligations `rc1, ..., rc4`, corresponding to **R.1, ..., R.4** in Definition 2.1.2) listed within the `RestrictionComb` class. Each of these proof obligations represents one of the

four informal rules a combinator must satisfy in order to be considered a restriction combinator. However, we would rarely ever need to specifically refer to the proofs of these four propositions — instead, we treat a restriction combinator as strictly a morphism of maps, both informally and formally (because of the coercion defined). Thus, we define a coercion wherever it exists informally. In this case, the coercion is as follows:

```
Coercion rc:  RestrictionComb >-> rcType.
```

With `RestrictionComb` defined in this way, it is natural to define `RestrictionCat` as a type class whose structure combines its two arguments, a category and a restriction combinator defined in that category, with no additional members or proof obligations.

Recall that Tot(C) is a subcategory of a restriction category with the same objects as the restriction category and only the total maps (see Definition 2.1.4). Note also that this type of subcategory is called a *wide subcategory* — that is, a subcategory which includes all the objects and only some of the maps of the larger category. To formalize total subcategories, we may build a term `Tot` as follows: `Tot` takes (is dependent on) two arguments, a category and a r.c. in that category, and returns a map from the corresponding restriction category into the `Category` type class.

```
Definition Tot (C: Category) (rc:  @RestrictionComb C):
RestrictionCat C rc → Category.
```

Note that here, even though the keyword `Definition` is used, we only give the header (to show its type and the arguments it takes) of the term we are about to define (we discuss the actual definition below). Note that the header is displayed using notation described earlier, by giving two arguments with their types before the colon, and presenting the rest of the header, which is dependent on these arguments, after the colon. Also, the type of the restriction combinator `rc` is dependent on the category `C` because `rc` must be defined in the given `C`. After the definition of `Tot` is

filled in, it will have the form:

```
fun (C:Category) ⇒ fun (rc:@RestrictionComb C) ⇒

    (fun D:RestrictionCat C rc) ⇒ ...
```

Note also that the '@' symbol above tells us that the type class it precedes expects a list of arguments with which it must be instantiated (all implicit as well as explicit arguments). It is used when Coq is not expecting the implicit arguments to be listed, but we want to list them explicitly. The process of building a term of this type (`Tot`) may be done by explicitly defining the total subcategory of the given category `Tot`, but fortunately the selected category theory library contains code that can be used to instantiate a wide subcategory directly, as well as a coercion from a wide subcategory to a category. Since C is a wide subcategory of C, we make use of these features.

We next describe how we fill in the definition of `Tot` using tactics, only showing some here in order to present the essence of a definition. Each time we build a subcategory of a particular category using the library we have selected, we must define a predicate on the objects of the larger category (for a full subcategory), a predicate on the maps (for a wide subcategory), and a predicate on both maps and objects when a category is neither full nor wide. To build a term of type `Category` that is a total subcategory of the given restriction category, we instantiate the `Wide_Subcategory` type class with the category `C` and the predicate representing the set of total maps with respect to the r.c. in this category, `TotMaps rc R: forall a b, Hom a b → Prop`. This predicate is true when the restriction of a given map `f: Hom a b` is equal to the identity.

The following code uses the tactic `apply` to match the current goal (in this case, of type `Category`, as we are defining a category containing only total maps), against the conclusion of the type of the term `(Wide_SubCategory C (TotMaps rc R))` — again, `Category`, and generates subgoals for each of the premises of this term. So, after the following tactic is used for defining a category,

```
apply (Wide_SubCategory C (TotMaps rc R))
```

two subgoals are generated (corresponding to the premises), which we must fulfill in order to complete the goal. These are the two proof obligations corresponding to the informal proofs we are required to do for proving a subcollection of objects and maps forms a subcategory:

i. prove the identity map is total

ii. prove composition of total maps is total

For the code we have written for the fulfillment of these obligations please refer to the code library (**Restriction.v**). To formalize additional categorical structure specific to restriction categories (outlined in Section 2.1), we must define type classes encapsulating these concepts. For example, a partial terminal object type class must include an object (which must be restriction terminal in the given category). It must also contain maps into this object, defined for each object in the category. And finally, this type class must contain the necessary proof obligations needed to show that the given object (considered together with the given maps) is restriction terminal.

```
Class ParTerm {RC: RestrictionCat}:  Type:=      {

     p_term:  RC;

     pt_morph:  forall (a:  Obj), Hom a p_term;

     morph_total:  forall (a:  Obj), rc _ _ (pt_morph a) = id a;

     id_is_ptm:  id p_term = pt_morph p_term ;

     pt_morph_unique_greatest:  forall (a b:  Obj),

     forall (f:  Hom a b),

     ((pt_morph b) o f) = (pt_morph a) o (rc _ _ f )

}.
```

The symbol '_' above is used in place of an argument which is expected to be given explicitly, but which the Coq system is able to infer in certain cases, such as the

one above. Recall also that `rc` comes from the `RestrictionComb` type class, and represents the combinator mapping itself (rather than the associated proof obligations). The types `Obj, Hom, id` and `composition`, on the other hand, come from the `Category` type class found in the library code. The type class corresponding to partial products (see Definition 2.2.1) is defined in a similar way, called `ParProd` in the code.

```
Class ParProd `{RC : RestrictionCat} (a b :  RC) : Type
```

This type class takes three arguments: a restriction category `RC`, and two objects `a` and `b` in this category. Given these three arguments, defining an instance of `ParProd RC a b` corresponds to (informally) defining a restriction product of `a` and `b`.

With the necessary cartesian restriction structure built over top of the existing category theory library, we may build the type class encapsulating this structure:

```
Class CartRestrictionCat `{C : Category}

`(rc :  @RestrictionComb C) `{RC : @RestrictionCat C rc} :  Type :=

{

     RCat_term :  ParTerm ;

     RCat_HP : Has_pProducts

}.
```

An instance of this type class requires a category `C` along with instances of a restriction combinator and a restriction category (the parameters `rc` and `RC` above), and an instance of the terminal object type (called `RCat_term`). It also requires defining a term of type `Has_pProducts`. A term of this type represents a collection of partial products of all pairs of objects in the given `RC`. That is, for any `a, b : RC`, we have `RCat_HP a b :  ParProd a b`, so that the header of `RCat_HP` is given by

```
Has_pProducts `{RC : RestrictionCat } := forall a b, ParProd a b.
```

With these definitions in place, we next demonstrate that a partial product constitutes a (true) product in the total subcategory of the given category. Formally, this can be expressed by a term of the following type, which defines, given a partial product of `a` and `b` in the larger category (which includes all maps), a (true) product in the `Tot` subcategory:

```
Definition defProdInTot `{C: Category}
`(rc:  @RestrictionComb C) `{RC: @RestrictionCat C rc} :
forall (a b:  TotR rc RC), forall (aXPb:  ParProd (proj1_sig a) (proj1_sig b)),
(Product (proj1_sig a) (sig1_proj b)).
```

Recall that term `proj1_sig a` for a sigma-type `a` returns the first term of the dependent pair. The term `TotR rc RC : RestrictionCat ...` formalizes the total subcategory of a given restriction category `RC`, itself considered as a restriction category. That is, it is defined to be an instance of the restriction category type class, and is made up of the objects of the category `C` within the given restriction category `RC : RestrictionCat rc C`, together with the restriction structure inherited from `RC`. However, in the case of `TotR rc RC` subcategory, we only consider the total maps of `RC`, so that the restriction of every map `f : Hom a b` in `TotR rc RC` is trivial (coincides with identity on `a`).

The type class `Product` seen in the `defProdInTot` definition above, as well as the type class `Terminal` in the `defTermInTot` definition below are the type classes, defined in Amin Timany's original category theory library (see [45] or the library code available on GitHub), which formalize cartesian structure (products and terminal objects, respectively) in a category. The definition `defProdInTot` uses the partial product identities, which correspond to (true) product identities in the case when we consider only the total maps in the category. The definition of a (true) terminal object in terms of a partial terminal object in the total map subcategory has also been formulated in a similar way. The header of this definition is below.

```
Definition defTermInTot ...  `{RC: @RestrictionCat C rc} :

forall t:  ParTerm , Terminal (TotR rc RC).
```

Note that in the code examples in this thesis, we use the symbol '...' a number of times. The parameters omitted are almost always the full list of the type classes needed to define the last parameter (`RC`, in the above definition). That is, the missing parameters would be the category `C` and the restriction combinator `rc` that make up the `RestrictionCat` type class. In many cases, these could be implicit and filled in by Coq, but the definition of trivial cartesian restriction structure has given Coq an alternative to fill in these arguments (if they were implicit) instead of the structure we expect to use (given by these explicit parameters). We will discuss this trivial restriction structure later in this section, along with more in-depth comparisons between the library-defined true products and terminal objects, and the restriction versions we have defined.

The definitions above indicate that the formalization of type class and coercion structures has so far been defined in a way that preserves the basic relationships (demonstrated between these objects informally) in a formal setting.

5.3 Formalizing Turing Structure

Finally, we have sufficient categorical structure formalized to define the constructs and propositions needed to formally describe the nature of Turing categories. To illustrate how this is done, the following is a definition of when a $T_{x,y}$ index is universal (which encodes Definition 2.3.1 in Coq).

```
Definition TxyUniv `{CRC: CartRestrictionCat}

(a x y:  RC) (aXx:  ParProd a x) (Txy:  Hom aXx y):  Prop:=

(forall z:  RC, forall zXx:  ParProd z x,

forall (f:  Hom zXx y), fAdmitsTxy a x y aXx Txy z zXx f).
```

Remark: Note above that the parameter `RC : RestrictionCat` is not given explicitly in the list of arguments in the `TxyUniv` definition. The reason for this is Coq's ability to guess implicit arguments, as well as the system of coercions defined throughout this formalization. Recall that parameter `'{CRC: CartRestrictionCat}` itself has a number arguments,

```
CRC: @CartRestrictionCat C rc RC
```

It is not necessary to list these type class arguments explicitly because Coq can match them to those in the argument list in the header of the type class `CartRestrictionCat`. In particular, the argument `RC : @RestrictionCat C rc` is the type of variables `a x y` in the `TxyUniv` definition, but is not explicitly listed. Note also that the coercions we have defined allow us to treat objects of types `C: Category, RC: RestrictionCat` and `CRC: CartRestrictionCat` as if they were all of type `Obj` in the underlying category `C` found in all of these type classes. Throughout this project (upcoming definitions included), each time we define a type class which consist of a category and some additional structure, we also define coercions in order to be able to treat terms whose type is an instance of this new class as objects in the underlying category of this type class. Then, in the code above, we could have included `(a x y : C)` or `(a x y : CRC)` instead of the `(a x y : RC)` parameter, and it would have made no difference in the resulting definition. However, it is important to note that in some cases, giving full lists of both explicit and implicit type class arguments is needed to avoid ambiguity, more on this in Section 5.5.1.

The definition of `fAdmitsTxy` is omitted, but it captures the content of part (i) of Definition 2.3.1. The predicate `TuringObj: Obj -> Prop` (omitting the relevant implicit arguments) is then defined using `TxyUniv`. Now, to build a Turing category, all that is needed is a type class that is instantiated with a cartesian restriction category and a Turing object in that category.

```
Class TuringCat {CRC: CartRestrictionCat} (A: Obj): Type:=
```

```
{ AisTuring:  TuringObj A }.
```

As in the informal definition of a Turing category, the • map need not be explicitly part of the `TuringCat` type class. Defining a • map explicitly would be required in the process of proving an object A is Turing because the existence of such a map must be demonstrated. However, for a particular Turing object, this map need not be unique. Once a • map with the necessary properties is defined, the $T_{x,y}$ maps for all other objects x, y must factor via this map in a specific way. Thus, they are automatically established upon specifying a particular application map.

5.4 Proving Results about Turing Categories

With the definitions in place, we can now prove various results about Turing categories. The first result to be proved about Turing categories in the original Turing categories paper is that every object in a Turing category is a retract of the Turing object (see Lemma 2.3.3). The informal proof, as many other proofs in category theory, is a commuting diagram, with a lot of obvious morphisms suppressed. Recall that in the formal language setting, however, every rule used in a proof must be either part of an automated strategy or applied by the user directly.

The following is the formalization of the definition of a retract, followed by the Coq statement of Lemma 2.3.3 — a key Turing category result.

```
Definition isRetractOf `{C: Category}:  C → C → Prop:=

fun (x y:  C) ⇒ (exists (m:  Hom x y) (r:  Hom y x), r ∘ m = id ).

Lemma everyObjIsRetract ...  `{CRC: @CartRestrictionCat ...} `(A:CRC)

`{T : @TuringCat ...  A}:  forall x:  T, isRetractOf x A.
```

Here as well as in subsequent lemma statements, we often omit (for readability reasons) implicit arguments which are given explicitly in our code. The proof of the

above lemma requires proofs of several auxiliary results about the nature of cartesian restriction structure, which we have also proved as part of this formalization, such as

i. $\overline{1} = 1$

ii. $X \times 1 \cong X \cong 1 \times X$ for partial products

iii. $\langle f, g\rangle h = \langle fh, gh\rangle$ for partial products whenever $r \circ m = 1$, $\overline{m} = 1$

For the complete proof code see the `everyObjIsRetract` lemma in the **Turing.v** file. The majority of proof code for the relevant lemmas consists of applications of rules about composition with identity and associativity, as well as the restriction rules. Note that in our formal proof of the above lemma (just as is done in the informal proof), we explicitly define an embedding-retraction pair for every object in the category into the Turing object; however, for any propositions proved about Turing categories, the fact that such a pair exists suffices.

It is worth noting that no additional automation has been implemented in this formalization — no hints or additional tactics have been defined. The reason for this is the chains of multiple coercions that have been defined, as well as the different options defined for the restriction structure. Because of this, any automation defined will be applicable only in a few specific situations, diminishing its usefulness as compared to building a particular proof directly.

In general, there is little repetition in the proofs and definitions of this formalization. Furthermore, each proof and definition is a study of the way the informal concepts can be made susceptible to formalization. Thus it is reasonable to manipulate the formalisms directly as part of the process of gaining a better understanding of them and the applicable techniques. However, built-in Coq proof automation can be helpful to avoid repetition of basic sequences of tactics, such as `intros; destruct; ...` etc. Such a sequence is useful in a situation where, for example, a previous tactic generates several proof obligations, each of which is of the form `forall a :`

`A, ...` , where `A` is a sigma-type. This is often the case for defining an instance of a type class. Here, the tactic `intro` formalizes forall-introduction. The tactic `destruct ⟨ term ⟩`, applied to a term of a dependent type, decomposes this term into terms of the types it is made up of, making it easier to manipulate and simplify the goal [18].

In order to formalize the next lemma, we must first consider how to implement restriction structure trivially in an arbitrary cartesian category. This can be done by defining $\overline{f} = 1$ for all maps f in the category. Note that R.1, ..., R.4 (see Definition 2.1.2) are immediately satisfied under this (trivial) definition. This definition also makes conventional products and the terminal object into restriction products and a restriction terminal object, respectively, which is easy to verify. Here is how to do this in Coq:

```
Definition triv_rc `{C: Category} :  rcType C :=

fun (a b:  C) (_:  Hom a b) ⇒ id a.

Instance triv_RC `{C: Category} :  RestrictionComb C.
```

The way `triv_rc` is defined coincides with the informal definition of the restriction $\overline{f}$ of every map $f : a \to b$ to be the identity on a for every f. The `Instance` keyword on the next line signifies the definition of an instance of a term of type `RestrictionComb C`. Defining an instance of a type class means supplying the terms that make up the type class (in this case, the mapping defined by `triv_rc`), then fulfilling the proof obligations (`rc1, ..., rc4` for the type class `RestrictionComb C`). The proof that this definition satisfies the required proof obligations is very straightforward. The idea of the formal proof strategy is as follows:

i. give `triv_rc` as the restriction combinator in the category,

ii. unfold and destruct all the relevant definitions and structures,

iii. for each of the four rules, complete a number of *rewrites* of the goal by using

the rules about composition with identity (which are part of the `Category` type class within the library)

Here, built-in Coq proof automation can be used to avoid repeating the code. The term *rewrites* refers to the replacement of one term with another of the same type when a proof of the equality of these terms is in the list of currently available hypotheses for the goal. The Coq tactic `unfold ⟨ term ⟩`, mentioned above, replaces `⟨ term ⟩` with its full definition as it is given elsewhere in the code.

Note that some of the formalization code included in this thesis (such as the two definitions below) requires a certain familiarity with Coq commands and syntax, and is intended to showcase (to the reader who is more familiar with the proof assistant) the details of the approach we are taking in the building of the relevant proofs and definitions.

The following definitions are of the partial terminal object and partial products in a category `C` where the restriction combinator is taken to be the trivial one, `triv_RC`.

```
Instance triv_ParTerm `{C: Category} `{T: @Terminal C}:  ParTerm.

destruct T. exists terminal t_morph.

...

Instance triv_Prods `{C: Category} `{HP: @Has_Products C} :

forall a b, ParProd a b.

destruct (HP a b).  exists product Pi_1 Pi_2 Prod_morph_ex.

...
```

Here, for brevity, we have omitted the code which completes the proof obligations required for building instances of the `ParTerm` and `ParProd` type classes (using '...'). Defining instances of these type classes is possible whenever the given category has (true) products and a (true) terminal object. This is why the instantiations take the parameters `T: Terminal` (an instance of the library-defined `Terminal` class) and `HP: Has_Products C`, a collection of (true) products for every pair of objects in the

category, also defined in the original library :

```
Definition Has_Products (C : Category) :  Type := forall a b, Product a b.
```

Note that we have modeled our `ParTerm` and `ParProd` type classes after the (true) terminal object (`Terminal`) and product (`Product`) type classes defined in the library we have chosen to build our formalization on. The terms `T` and `HP` coincide with their partial counterparts exactly when we take the restriction combinator to be trivial, as all the required proof obligations simplify to the proof obligations of the original library classes `Terminal` and `Has_Products`. The tactic `exists`, followed by the terms `terminal, t_morph` (for `triv_ParTerm`) and `product, Pi_1, Pi_2, Prod_morph_ex` (for `triv_Prods`) instantiates the classes. These terms are the (true) product counterparts to the terms contained in the `ParProd` and `ParTerm` type classes, i.e. `p_term, pt_morph` for the terminal object, and `p_prod, Pi_1p, Pi_2, pProd_morph_ex` for products, respectively. That is, in the case of the trivial combinator, these coincide for restriction and true cartesian structure.

The above instantiations are not complete: Coq also expects the associated proof obligations (the formal version of showing the appropriate diagrams commute, see Diagram 2.2.1). These proofs are omitted here but can be found in the code file (**Turing.v**). Finally, we instantiate a cartesian restriction category `triv_CRCat` with the instances of the defined restriction combinator class, the (restriction) terminal object class, and the (restriction) products class. Here, `CCC` is the (original) library-defined type class encapsulating cartesian closed category structure

```
Instance triv_RCat `{C: Category} `{aCCC: CCC}:  RestrictionCat aCCC triv_RC.

Instance triv_CRCat  `{C: Category} `{T: @Terminal C}

`{HP: @Has_Products C}:  CartRestrictionCat triv_RC.

exists.  exact triv_ParTerm.  exact triv_Prods.
```

Next, we have formalized a proposition concerning the relationship between cartesian closed and Turing structure: in a CCC C which contains an object A such that

every object is a retract of A, this object A is a Turing object if we equip C with trivial restriction structure. We have,

```
Theorem aTuringCCC `{C: Category} `{aCCC: @CCC C}:

forall A: (triv_CRCat aCCC CCC_term CCC_HP),

(forall b:  (triv_CRCat aCCC CCC_term CCC_HP), isRetractOf b A )

-> TuringObj A.
```

In this theorem, we use the terms making up the cartesian closed structure inside the `CCC` type class, found in the original library. In this class, there are terms `CCC_term : Terminal C` and `CCC_HP : Has_Products C` which give cartesian structure (also, `CCC_HEXP : Has_Exponentials C` for exponential structure, which we do not see in the definition of the theorem, but is used in the proof).

Note Coq is able to treat A as an object in the underlying category C as well as an object of any of the instances of categories built from C by introducing additional structure. This is due to all the necessary coercions being in place. Here, the object A is considered Turing in the `triv_CRCat` cartesian restriction category when the corresponding cartesian closed category meets the 'every object is a retract' condition (see Example 2.4.6).

At this point it is worth pointing out that a cartesian closed category does not necessarily have to have trivial restriction structure. It may be possible to express a more general (restriction) version of closedness in a cartesian restriction category in a meaningful way, using the existing restriction structure (rather than imposing trivial restriction structure), see the cartesian closed restriction categories discussion in [14], Section 2.3. This could be the subject of future formalization work.

The following are other results from the theory of Turing categories (found in [Cockett and Hofstra], [15]) that we have formalized. Our selection includes the results the authors have found most interesting and representative of the nature of computation in Turing categories, as well as those best suited for testing the correctness of

the formal categorical definitions.

i. An object B in a Turing category with Turing object A is Turing if and only if it is a retract of A (this is Lemma 3.5 in [15], and Lemma 2.3.4 in this thesis). In our code,

```
Lemma ARetOfeveryTO ... '(A: Obj) '{T: @TuringCat ... A} :
forall A': T, @TuringObj ... A' ↔ isRetractOf A A'.
```

ii. The halting domain is m-complete (this is a version of Corollary 3.9 in [15], Lemma 2.3.8 in this thesis). Recall that the halting set in the case of a Turing category T is simply $\overline{\bullet}$. In our code, it formalized by the lemma

```
Lemma halting_m_comp ... '(A: CRC) '{ T: @TuringCat ... A } :
forall (X: T) (bullet: Hom (RCat_HP A A) A)
(tm: @TuringMorph ... bullet ) (e: Hom X X) ,
e = (rc _ _ e) → (exists (f: Hom X (RCat_HP A A)), ((rc _ _ f ) = (id X)) ∧
(rc _ _ e) = (rc _ _ ((rc _ _ bullet) ∘ f))).
```

Note that the predicate `TuringMorph`, above, when applied to a map of type `Hom (RCat_HP A A) A`, is true whenever the given map is a Turing morphism (see Definition 2.3.2).

iii. An equivalent characterization of Turing categories in terms of the Turing morphism and object embeddings (this is Theorem 3.4 in [15], and Theorem 2.3.6 in this thesis). In the code, it is

```
Definition eq_charac ... '(CRC: CartRestrictionCat ... ) (A: CRC) :
(exists (bullet: Hom (RCat_HP A A) A) , CompMorph ... CRC A bullet) →
(forall a, @isRetractOf C a A) → @TuringObj ... CRC A.
```

The predicate `CompMorph` evaluates to true for a given map `bullet : (RCat_HP A A) A` whenever `bullet` computes all maps of type `f : Hom (RCat_HP A`

`A) A`, i.e. is a universal application (see, again, Definition 2.3.2). More specifically, when there exists a factorization of the map `f` via the `bullet` map for some total map `h : Hom A A`, as required in Definition 2.3.1 of Turing structure.

All of these proofs are found in the **Turing.v** file. Completing these proofs provides solid evidence that the selected model for abstract computation can be encoded and reasoned about using a programming language built on the basis of a traditional theory of computation.

5.5 Formalizing Combinatory Complete Structure and the Turing Category Embedding Result

Next, we move on to formalizing the nature of computation in a Turing category, as well as a category-theoretic description of the relationship between categories built from just the computable maps between m- and n-fold products of the Turing object with itself. Note that the proofs of certain propositions in this section have been omitted (by using '`Admitted`' in the code), which means that the formal proofs are not complete. For example, we completed the proofs of `rc3` and omitted the proofs of `rc1, rc2` and `rc4` for the Karoubi envelope of a category (see Definition 2.1.6) because they follow a strategy very similar to other proofs that have been completed. We have also omitted certain well-known results derived prior to the development of Turing category theory (e.g. the proof of the two-way implication between existence of k, s combinators and combinatory completeness, Lemma 2.4.5).

5.5.1 Full Cartesian Restriction Subcategory of Objects A^n

We begin by formalizing the category of polynomial morphisms on an object A in a restriction category C. That is, all maps of the form $A^n \to A^n$. In order to be able to

treat maps in this category as maps in the larger category C, we must instantiate such a category as a (full) subcategory of C, with n-fold powers of A as objects. The n-fold products of an object A are defined using a fixpoint operator that builds a partial product $A \times (A^{n-1})$ at each iteration. The fixpoint operator is a special syntax that Coq provides for generic primitive recursion (for the definition of primitive recursion, see Definition 6.2.1).

Next, we use a structure to encapsulate the dual nature of an n-fold product: such a structure must behave as a terminal object for $n = 0$, and as a partial product for $1 \leq n$. The resulting construct is built as a fixpoint definition, `nthProdC`. The term `nthProdC C A n` corresponds to A^n in the category C. The code described in this subsection is found in the **CompA.v** library file, and **Restriction.v**.

In the library selected for this formalization, defining a full subcategory requires first defining a predicate `P: C → Prop`, where for each object `X: C` that is also in the subcategory, the resulting proposition `P X` is provable. Now, for a given object `X: C`, we have chosen to express this predicate in terms of the existence of an (n-fold) partial product object because it seemed to be a natural choice, translated as directly as possible from an informal definition: `P X := exists (n: nat), X = nthProdC C A n`.

Remark: This is the first instance in this formalization project where we are faced with a decision of whether to add a (type-specific) version of the axiom of choice: we would like to be able to *choose* a specific `n` for which `X = nthProdC C A n` so that we are able to reason about `X` as an n-fold product. In the **CompA.v** file, however, we reason only about structure inherited from the larger category containing `A` — the restriction combinator, products and the terminal object. Therefore, the proofs of the necessary propositions are also inherited, and we do not require reasoning about objects specifically as n-fold products. We note that the above-mentioned version of the axiom of choice will be required for reasoning about computability of maps in the following subcategory: the full subcategory of a given cartesian restriction category

C with objects of the form A^n for some A : C, which corresponds to the category informally denoted Comp(A). We will discuss this in more detail in the next chapter in the process of formalizing Comp($\mathbb{N}$).

Given `CRC : CartRestrictionCat` and `A :CRC`, the term `all_prod_maps_cat CRC A : Category` represents the category Comp(A) in our code. As we have done when defining other subcategories in our formalization, we apply a (library-defined) term `Full_SubCategory` which builds a full subcategory of a given cartesian restriction category `CRC`. To build this full subcategory we use the predicate `P` on the objects of `CRC` to differentiate which ones are contained in the category (those of the form A^n) and which are not. Next, we discuss defining cartesian restriction structure in this category to build a cartesian restriction category.

The restriction combinator in the category `all_prod_maps_cat CRC A` we have just defined is inherited from the underlying category. Since we are working with a full subcategory, it contains all idempotents (found in the larger category), on all its objects, required to define restriction structure. The proof obligations are again inherited because they hold in the larger category. We define the restriction combinator (we give only the header here):

```
Definition rcCompA '(CRC: CartRestrictionCat ) (A: CRC):

RestrictionComb (all_prod_maps_cat CRC A).
```

We discuss below why this `RestrictionComb` is declared as a definition and how this fits into the schema of formalizing Comp(A), the cartesian restriction subcategory of a given `CRC`. Another point to consider when defining this type of subcategory is the difficulty of manipulating partial products when they differ by associativity. That is, comparing $A \times A^n$ and $A^n \times A$, for instance, will require a rather complicated (fixpoint) definition, accompanied by the necessary proofs about the structure relating the objects and maps in the two instances of the `ParProd` type class (i.e. $A \times A^n$ and $A^n \times A$ in this case).

We have chosen to implement a shortcut for ease of working with partial products: we have equated the objects in this subcategory in the following way: $A^n \times A^m = A^{n+m}$ instead of defining a separate category *mod* this congruence relation (or any other more meticulous alternative). This is formalized by the axiom `AnAm_Anm_pf`. We leave this as future work to build a formal proof of this equality. Products in the subcategory we are building can now be defined using this equality, and will correspond exactly to the product definition in the underlying category (so, can be treated as if inherited from C). Thus, the subcategory described above is defined to be a cartesian restriction category for any underlying cartesian restriction category C.

Universe Polymorphism and Type Classes. Now, recall the type hierarchy described in Chapter 4.1. Definitions and instances of type classes must obey this hierarchy, although they may be instantiated with objects that reside at different levels in the hierarchy. This concept is known as *universe polymorphism of definitions* (see [18], Chapter 29). It is worth pointing out that in the process of defining the subcategory described above, followed by the restriction combinator and products, and then the associated cartesian and CR categories using these structures, it has been a challenge to organize all these into *definitions* and *instances* in a way that avoids universe-polymorphic conflicts during instantiation. The conditions that would result in a conflict are formalized in Coq's type-checking algorithm. These conditions are a collection of rules about terms of what types (including different levels of types of types, i.e. `Type : Type`) can be used to instantiate other types. In Coq 8.5, the universe polymorphic typing rules are more flexible due to binding a domain universe level at the definition level instead of making it global. For a detailed discussion, see [18], Chapter 29.

The full subcategory `all_prod_maps_cat CRC A` (of all objects A^n, for a particular cartesian restriction category `CRC` and object `A` in that category) is defined as an *instance* of a `Category`. The reason for this is that given an instance of a larger

category, i.e. `CRC : CartRestrictionCat ...`, we are building another instance of a category. In this case, the specifics of what exactly the subcategory is are given in the process of defining this instance.

Note, however, that we have defined a restriction combinator in this category as a definition, i.e. `Definition rcCompA` above. This definition sets the r.c. equal to that in the underlying cartesian restriction category (i.e. the `RCat_RC : RestrictionComb C` term of the implicit (type class instance) parameter `RC : @RestrictionCat C ...`), and `RCat_RC` also contains the associated proof obligation terms.

Remark : In general, if we demand that a structure found in a subcategory (e.g. restriction structure, cartesian structure) be inherited from a larger category, we must give a `Definition` of this structure, rather than an `Instance`, wherein we explicitly indicate that it is inherited. This allows the proofs of the proof obligations to be inherited. We cannot define these proofs in the general case. Since the category which these structures must be inherited from is a parameter (so we are defining the structure in the general case for any given `CRC`), we do not know the nature of the structure and cannot complete the proof obligations any other way.

Attempting to build an `Instance` of structure in the way discussed above (via inheritance) gives a universe polymorphic error in Coq. Coq does not allow building an instance of particular structure with (unknown, i.e. found in the terms in type class instances listed as premises) proofs and terms in a given parameter. However, it is possible, given a parameter, to set the structure we are defining *equal* to the structure found in the given parameter. That is, both of these structures must exist on the same type level. Doing this using `Instance` puts them on different type levels.

In general, given the type classes we have defined in the **Restriction.v** file, there is a unique way to structure the type-level hierarchy of definitions and instantiations of all the necessary structures (the subcategory, its restriction combinator, corresponding restriction category and cartesian restriction category, products and

terminal object) into a cohesive collection that does not violate the universe polymorphism conditions when instantiated with a particular category, object A, partial products, r.c., etc.

The following is the organization of instances and definitions of the categorical formalisms needed to build the Comp(A) cartesian restriction category (we only give the headers in each case). Note that we have made several attempts at using a different configuration of `Instance` and `Definition` keywords to build these categorical structures, but they have resulted in typing errors.

(i) Create an instance of the full subcategory of n-fold products of an object `A`

```
Instance all_prod_maps_cat (C: Category) (rco:  @RestrictionComb C)
(RC: @RestrictionCat C rco) (CRC: @CartRestrictionCat C rco RC) (A: CRC): Category.
```

(ii) Define the restriction combinator in the resulting category

```
Definition rcCompA ...  (CRC: @CartRestrictionCat ...)
(A: CRC) : RestrictionComb (all_prod_maps_cat ...  CRC A).
```

(iii) Create an instance of a restriction category with this restriction combinator definition

```
Instance all_prod_maps_Rcat ...:
RestrictionCat (all_prod_maps_cat ...)  (rcCompA ...)  .
```

(iv) Next, given two objects `a` and `b` define *object* corresponding to the product of `a` and `b` in the restriction category defined in (iii)

```
Definition CompAprod ...:  forall a b , (all_prod_maps_Rcat ...).
```

(v) Now define the term of type `ParProd a b`, using the product object as in (iv)

```
Definition CompA_Prods ...:  forall a b , ParProd a b.
```

(vi) We also *define* a term of type **ParTerm**, called `CompA_Term`, which is the terminal object in the `all_prod_maps_Rcat ...` category.

(vii) The resulting cartesian restriction subcategory is instantiated as follows

```
Instance CompA_CRCat `(C: Category) `(rco:  @RestrictionComb C)
`(RC: @RestrictionCat C rco) `(CRC: @CartRestrictionCat C rco RC) (A: CRC):
CartRestrictionCat (all_prod_maps_Rcat C rco RC CRC A) (rcCompA C rco RC CRC A)
(all_prod_maps_Rcat C rco RC CRC A).
exists.  exact (CompA_Term C rco RC CRC A). exact (CompA_Prods C rco RC CRC A).
```

Now, let us compare this to defining cartesian restriction structure with a trivial restriction combinator (see Section 5.4). In that case, too, we had to organize instances and definitions to conform to a type hierarchy. Recall that we defined `Instances` of a restriction combinator, terminal object and products (given a category `C` as a parameter). Those formalisms also satisfied Coq's type-checking algorithm. The reason for that is that in the trivial restriction combinator case, we are not given an instance `rco : @RestrictionComb C` that we then show behaves like a restriction combinator in a smaller category (i.e. show it is inherited), but rather we define an entirely new instance of a restriction combinator. That is, we explicitly define the restriction of each map to be the identity, and then complete the necessary proof obligations. In a specific category, for example, Par, this would give a combinator that defines $\overline{f}(x)$ for each f and each x to be x.

Note that essentially all the parameters in the above `Instance` and `Definition` headers are given explicitly. The cases (i) - (vii) above are examples where Coq does not have enough information to fill them in automatically. For example, if we do not give the restriction combinator explicitly in the list of the arguments, Coq always guesses the r.c. to be the trivial r.c. instead of the one we have included in the list of parameters of the type definition. The symbol `...` is used in place of the omitted arguments for brevity.

5.5.2 PCAs and Combinatory Completeness

Having outlined the 'recipe' to obtain a (full) cartesian restriction subcategory of all n-fold products of an object A (with polynomial maps) in a CRC, we may now move on to formalizing PCA structure and combinatory completeness in such a category. Recall (see Definition 2.4.1) that a PCA in a cartesian restriction category C is made up of an object $A : \mathsf{C}$ and a map $\bullet : A \times A \to A$, as well as the computability conditions on the polynomial morphisms $A^n \to A^m$ (that is, factorization via a total map and $\bullet$). We begin by defining a single application `bullet_once` of `bullet` (informally, $\bullet$) to the first two arguments of the product, $A \times A \times A^{n-2} \to A \times A^{n-2}$. Then, we define the n-fold `bullet_n` application using the operator,

```
Fixpoint bullet_n ...  (A: Obj) (n:  nat)
(bullet:  Hom (RCat_HP A A) A):
Hom ( RCat_HP A (nthProdC rc A n)) A.
```

which decreases on argument n and returns $\bullet$ for $n = 1$ and $\bullet\Delta$ for $n = 0$ (see discussion in Section 2.4 before Definition 2.4.1). Also, note that this definition requires the Coq `Fixpoint` operator for primitive recursion, but we do not show the body of the definition.

Next, we must formally characterize when a polynomial map $A^n \to A^m$ is computable (see Definition 2.4.1). The case for $A^n \to 1$ is different than $A^n \to A$, so there are separate propositions defined that are provable whenever each of these two cases is computable,

($A^n \to A$): `Definition f_comp (CRC: CartRestrictionCat)`

```
(A: CRC) (bullet:  Hom (RCat_HP A A) A) (n:  nat)
(f:  (Hom (nthProdC rco A n) (nthProdC rco A 1))):  Prop.
```

($A^n \to 1$): `Definition isAppStructTerm (CRC: CartRestrictionCat)`

```
(A: CRC) (bullet:  Hom (RCat_HP A A) A) (n:  nat)
```

```
(f: (Hom (nthProdC rco A n) (nthProdC rco A 0))): Prop.
```

Recall that `RCat_HP A A` is the partial product of `A` with itself in the given `CRC`. Now, having defined the $A^n \to 1$ and $A^n \to A$ base cases, we can define the computability predicate for a polynomial map $A^n \to A^m$ for an arbitrary m by

```
Fixpoint isAppStructFornProd (A: CRC) (bullet: Hom (RCat_HP A A) A)
(n m: nat) (f: (Hom (nthProdC rco A n) (nthProdC rco A m))) : Prop:=
match m with
      | 0 => (isAppStructFornProd_test_m A bullet n m f (isAppStructFornProd) )
      | S m' => (isAppStructFornProd_test_m A bullet n m f (isAppStructFornProd) )
end.
```

Note that the notation `match _ with | ... end` in Coq is used for pattern matching. In the definition above, whenever the parameter `m` is equal to 0, the resulting proposition is the term that follows the `0 =>` arrow, and whenever for some `m' : nat`, `m = S m'`, the resulting proposition follows `S m' =>`. The notation `S m'` in the pattern matching clause above is a way to represent the natural number $m' + 1$ in Coq, as `S : nat -> nat` is a constructor which gives successor of a given natural number. The type `nat` has only one other constructor, `0: nat`. The recursive call in the above `Fixpoint` definition is done by supplying the `isAppStructFornProd` predicate as one of the arguments to the predicate `isAppStructFornProd_test_m` inside the body of the definition. Now, we chose to build the predicate `isAppStructFornProd` using this auxiliary predicate of type

```
isAppStructFornProd_test_m `{C: Category} `{rco: @RestrictionComb C}
`{RC: @RestrictionCat C rco} `{CRC: @CartRestrictionCat C rco RC} (A: CRC)
(bullet: Hom (RCat_HP A A) A) (n m: nat)
(f: (Hom (nthProdC rco A n) (nthProdC rco A m)))
(test_prop: forall (A: CRC), Hom (RCat_HP A A) A → forall n m: nat,
Hom (nthProdC rco A n) (nthProdC rco A m) -> Prop): Prop. .
```

This predicate takes the same arguments passed to the `Fixpoint isAppStructFornProd` as well as the a predicate `test_prop` of the same type as `isAppStructFornProd`. Within the definition of this predicate, we perform case analysis on `m` as the cases for $m = 0, m = 1$ and $m \geq 2$ must be dealt with separately (see discussion below):

(`m=0,1:`) the predicate `f_comp` determines when a map into `nthProdC rco A 1` (informally, into A) is computable, and `isAppStructTerm` determines when a map into the terminal object is computable

(`m≥2`) in this case, we build the proposition as a conjunction of
— `f_comp A bullet n` applied to the composition of `f` with projection onto the first coordinate of `nthProdC rco A m`, and
— the predicate `isAppStructFornProd A bullet n (m-1)` applied to the map `f` composed with projection onto `nthProdC rco A (m-1)`

In our code, the term `isAppStructFornProd_test_m`, which appears in the definition of the recursively defined predicate `isAppStructFornProd`, is applied to the predicate itself as one of the arguments (the `test_prop` argument). Then, in the `m≥2` case above, a recursive call is made by the auxiliary predicate, as `isAppStructFornProd` is called (with `m` decremented by 1 this time).

It is a very useful feature of Coq that it is able to discern that this auxiliary predicate in fact decreases the argument m each time it is called by the fixpoint predicate. That is, the fixpoint definition makes an indirect recursive call, as in (ii) above, via invoking the auxiliary predicate on the (decremented) argument $m-1 < m$ (along with all the other arguments). The reason $m = 0$ and $m = 1$ are separate cases is that we must consider the computability of the *restriction* $\overline{f}$ of a map $f : A^n \to 1$ (that is, when $m = 0$), and $m = 1$ is the ordinary base case, where the computability of a map $A^n \to A$ is defined directly, see Definition 2.4.1.

Next, we define a predicate `AppSysIsCombComp` which takes as arguments a cartesian restriction category `CRC`, an object `A : CRC` and a map `bullet : Hom (RCat_HP A A) A`. Given these arguments, we define the resulting proposition in terms of `isAppStructFornProd`. It expresses that for all `n, m : nat` and all maps

```
f : Hom (nthProdC rco A n) (nthProdC rco A m)
```

the proposition

```
isAppStructFornProd A bullet n m f
```

holds. We may now give two formal characterizations of PCAs. Recall (again, from Definition 2.4.1) that an applicative system is simply a pair `A : CRC` and a map `bullet : Hom (RCat_HP A A) A`. An applicative system with object `A : CRC` and applicative map `bullet` is a PCA whenever either one of the following formal propositions is true :

`AppSysIsCombComp ...` `CRC A bullet`: this proposition expresses that the given applicative system is combinatory complete, or

`has_k_s ...` `CRC A bullet` : this proposition expresses that the given applicative system has k, s combinators .

In the definition of the `has_k_s`, the k, s combinators are *point maps* into `A` (as they are in the informal sense). That is, they have the type `@point ... A`, where `@point ... A` is a dependent pair combining a term `p : Hom RCat_term A` and a proposition that `p` is total. Finally,

`k_s_comb_comp_allP ...` `(A: CRC) (bullet: Hom (RCat_HP A A) A)`

is the formalization of Lemma 2.4.5, which relates the two formalisms above: an applicative system $(A, \bullet)$ in a given cartesian restriction category `CRC` has k, s combinators whenever it is combinatory complete .

5.5.3 Formal Idempotent Splitting and the Embedding

The final component to formalizing the characterization of Turing structure in terms of the underlying PCA (see Lemma 2.4.3) is to define the larger category into which a Turing category embeds. This category is defined in terms of the n-fold products of the Turing object, formally split idempotents on those objects, and the Turing morphism. Recall the idempotent splitting recipe (see Definition 2.1.6). Our formalization will follow this definition as closely as possible.

We begin by defining objects and maps in this category. Given a category `C`, the objects in its split-idempotent (also called split) category, called `split_obj` in our code, are dependent pairs consisting of an object in the original category and an idempotent contained in a given class E of idempotents. In the code, a class E of idempotents is represented by a predicate, provable when an idempotent (informally) belongs to the given class. So, the parameter representing a class of idempotents has the header

```
Definition idem_class `(C : Category) := forall (a:  C) (e:  Hom a a), Prop .
```

Given a predicate `E : idem_class` representing a collection of idempotents, we must define the split category objects as a (dependent) type pair:

```
Definition split_obj`(C : Category) (E : idem_class C) :
{ a:  C & {e:  Hom a a | ((E a e) ∧ ( e ∘ e = e )) } } } .
```

The `{ ... }` notation above represents the informal notion of a subset (or a set of dependent pairs). Inside the outermost { }, the symbol `&` is used to construct a set of dependent pairs, where the first term is an object `a` of a category C, and the second term is (informally) a member of a particular subset of the type `Hom a a`. Specifically, the second term (i.e. the term represented by the inner pair {}) is a subset of terms of the type `e : Hom a a` such that `e` is an idempotent on `a` and `E a e` is true (informally, e is in the class E).

We define maps in this formalization of a split category to also be dependent

types (i.e., informally, subsets of terms satisfying the property after the `'|'` symbol),

```
Definition split_hom '(C: Category) (E: idem_class C):=

fun ( ce df:  split_obj C E) => { f_ef:  Hom (projT1 ce) (projT1 df)

| (proj1_sig (projT2 df)) ∘ f_ef = f_ef ∧ f_ef = f_ef ∘ (proj1_sig (projT2 ce)) }.
```

Here, the terms `projT1` and `projT2` are projections to the first and second terms of a dependent pair, and `proj1_sig` is the projection onto the first term of a sigma type. For any two objects `ce df : split_obj C E`, the terms of type `f : split_hom ce df` are again pairs. The type of the first term, `Hom (projT1 ce) (projT1 df)`, corresponds to a map in the underlying category between the first projection of `cf` and the first projection of `df` (which are objects in the underlying category). The second term is a proposition formalizing the requirements the given map (in the underlying category) must satisfy in order for the term of the resulting sigma type (a pair consisting of a map and a proof of this proposition) to be of the `split_hom E ce df` type. These requirements come from the idempotent splitting definition (see Definition 2.1.6). Recall, also, from Section 4.3, that ∘ denotes composition of maps.

Next, we define composition and the identity maps in the category Split(C). Some of the proofs required to instantiate this collection of objects `split_obj` and morphisms `split_hom` as a category have been omitted from the code (using `Admitted`) because the informal versions of these proofs are immediate. We call the resulting category

```
Definition SplitC '(C : Category) (E : idem_class C) : Category.
```

in our code. In order to instantiate our formalization of Split(C) as a cartesian restriction category, we have defined a restriction combinator (along with completing proof obligations `rc1, ..., rc4`), and we give the header here:

```
Definition Split_rc (RC: RestrictionCat) (E: idem_class RC):

RestrictionComb (SplitC RC E).
```

Note again, here, that we are building a `Definition`, as we intend to use the definitions and proofs inherited from the given restriction category `RC`. This combinator is consistent with the restriction combinator in C on maps between objects of the form $(A, 1_A)$, but is a function also of e_A and e_B for a map $f : (A, e_A) \to (B, e_B)$ (see Definition 2.1.6 for details). We use the category `SplitC` and the restriction combinator `Split_rc` to define an instance of a restriction category, which we call `SplitRC`.

Next, we need to build cartesian restriction structure for `SplitC C E`. In order for the category Split(C) to have partial products, the collection of idempotents E must be closed under taking the product $e_A \times e_B = \langle e_A\pi_1, e_B\pi_2\rangle$ of any two idempotents e_A, e_B in E. Moreover, the identity on the terminal object must also be contained in the class of idempotents E. We state this as an (informal) proposition,

Proposition 5.5.1 Suppose C is a cartesian restriction category and E is a collection of idempotents in C. Then, the category Split(C) is a cartesian restriction category if the following hold:

(i) for any two idempotents e_A, e_B in E, their product $e_A \times e_B$ is also in E, and

(ii) the identity map on the terminal object in C is contained in E

Note that we have not formalized any counterexamples to the negation of the converse of the above proposition (thus, we have only proved the forward implication in our formalization). That is, we have built a formal interpretation of a split cartesian restriction category Split(C) with the assumptions that (i) and (ii) are true. We include proofs of propositions formalizing statements (i) and (ii) in the list of arguments of the term which defines an instance of a cartesian restriction category which is the restriction category `SplitRC` with cartesian restriction structure inherited from underlying cartesian restriction category `CRC`. These arguments and their types, corresponding to (i) and (ii) above, are as follows

(i) `pfe_cl : E_closed_prod ... CRC E`

(ii) `Ehas_id : E RCat_term (id RCat_term)`

We may now define an instance of the split cartesian restriction category,

```
Instance SplitCRC ... (E : ... ) (prf_cl : ... ) (Ehas_id ... ) :
CartRestrictionCat ... (SplitRC ... E).
```

The class of split idempotents that is considered in the result about the embeddings between Comp(𝔸), T and Split(Comp(𝔸)) (Lemma 2.4.3) is the entire collection of idempotents on all objects in the underlying category. Hence, we now define the category Split(Comp(𝔸)) as a cartesian restriction category with all idempotents split (and therefore, E provably meets the required closure conditions formalized by `Ehas_id` and `E_closed_prod`):

```
Definition SplitCompA_justCat_allIdem (CRC: CartRestrictionCat) (A: CRC):=
SplitCRC (CompA_CRCat CRC A) (rcCompA CRC A) (all_split ...) (asp ...) (h_id ...).
```

Here, `all_split` is the predicate that is true of all idempotents on all objects in `CRC`, as we want all idempotents to split in this category. The terms `asp` and `h_id` are proofs of closure under the `E_closed_prod` and `Ehas_id` closure conditions of the idempotent collection. They are immediate when all idempotents split.

The term `SplitCompA_justCat_allIdem CRC A` is the formal description of the Karoubi envelope (with all idempotents split) of the full subcategory of a cartesian restriction category `CRC` containing all objects of the form A^n, informally denoted Split(Comp(A)). Such a category can be defined for any cartesian restriction category `CRC`.

Now, recall (Lemma 2.3.3) that in a Turing category, every object is a retract of the Turing object. For this reason, in the case when `T` is in fact a Turing category, we may define the embedding of `T` into Split(Comp(𝔸)), as in Lemma 2.4.3. Note that for a Turing category with Turing object A, we use the notation Split(Comp(𝔸)) to represent combinatory completeness of an applicative structure in this category for

any Turing morphism. Informally, if we take the Comp($\mathbb{A}$) subcategory and formally split all idempotents, we can map every object X of T to the corresponding pair $(A, m_X r_X)$ in the splitting of the subcategory.

The formalization of this embedding is found in **Turing.v**. Defining the embedding Comp($\mathbb{A}$) $\hookrightarrow \mathsf{T}$ requires first defining a mapping on objects in T, which we formally define by the term `Comp_tur_o: @Obj (CompA_CRCat CRC ...) → @Obj T`, then a mapping on the morphisms in the category, `Comp_tur_m`. We use these terms to build an instance of the (library) type class `Functor (C C' : Category) : Type` (definition omitted here), and we also complete the required proof obligations. This instance corresponds to the informal embedding of Comp($\mathbb{A}$) into T. To formalize the notion that we have defined a functor that is an embedding, we build an instance of the (library) `Embedding` type class,

```
Instance Comp_T_Embedding ... (A: CRC) ( T: TuringCat rco A ):
Embedding (CompA_CRCat CRC ... A) T .
```

which we then instantiate with the `Functor` type class. We also complete the proof obligations to show that this functor is full and faithful (see the code for details). To show the second part of the Turing category embedding result, $\mathsf{T} \hookrightarrow$ Split(Comp($\mathbb{A}$)), we follow the same strategy. However, in order to define the functor on the category's objects and morphisms, we actually require the specific embeddings of each object into an object in Comp($\mathbb{A}$), or the corresponding idempotents. Recall that in a Turing category, every object is a retract of the Turing object A. This is provable from the *existence* of a morphism $\bullet$ satisfying certain properties.

Now, in the proof of an arbitrary proposition about a Turing category with Turing object A, we can always pick a specific embedding-retraction pair for an object X in the category when we know that such a pair exists — this is permitted in the logic of Coq. However, as in the case with selecting a specific `n` as the power of `A` for an object of the form `nthProdC rco A n` in the `CompA_CRCat _ _ _` category, we cannot pick

a specific embedding-retraction based solely on the existential statement (`exists m r, ...`) when building a term of non-propositional type, i.e. the embedding, in this case.

At this point, we could have added the axiom of choice (abbreviated AC), specifically allowing us to pick the embedding retraction pair when such a pair exists (there is no benefit to adding a more general version of the axiom of choice, but there is the drawback of having to consider the potential unintended consequences, i.e. the possibility of introducing inconsistency into the system). However, instead of adding AC, we decided to instead add the following parameter (in addition to the underlying cartesian restriction category `CRC`, the Turing category `T`, the Turing object `A : CRC`, etc.) to the header of the definition of the embedding functor:

```
emb_col: forall x: T, { mrx: (Hom x A)*( Hom A x ) |
((snd mrx) o (fst mrx)) = id x }
```

A term of type `()*()` is a pair of objects of the types indicated in the brackets. This parameter represents a collection of embedding-retraction pairs, one pair `mrx = (fst mrx, snd mrx)` for each object x in the category, with the first coordinate being the embedding, and the second - the retraction. We use the embedding-retraction pairs given by this collection in the definition of the `Functor` and `Embedding` type classes.

We have chosen this approach (rather than AC) in this case because defining embedding-retraction pairs for every object (into the Turing object) in a Turing category is possible as a result of the nature of Turing structure. A specific Turing structure is not part of the `TuringCat` type class, only the proof of its existence is, for the selected Turing object (i.e. the proof obligation `AisTuring`). When instantiating this type class, however, we must demonstrate the existence of Turing structure by defining it explicitly. Thus, for every instance of a Turing category we define, we would have the explicit collection of embedding retraction pairs needed for instan-

tiating the proof of part (ii) of Lemma 2.4.3 for this specific category. In contrast, given an object A in a category C, for an object $B \in \mathsf{C}$, if there exists an n such that $B = A^n$, it is never possible to choose the specific n outside the context of proving a proposition, not even when reasoning about a specific category C (this is discussed in more detail in 6.2.2).

Note that, for a particular Turing category T with Turing object A, we can prove the proposition that such a functor exists, since in the context of proving a proposition, we may explicitly pick the collection of embedding-retraction pairs for each object (because we have formally proved that such a collection exists in every Turing category, see `everyObjisRetract` in our code, see Section 5.4). Finally, we can define the second embedding required for the proof of Lemma 2.4.3. Instantiating an `Embedding` again requires supplying an object of type `Functor`, defined in our code as `Sp_Comp_T_Fuct`, and proving that it is both full and faithful. For the embedding of T into Split(Comp$\mathbb{A}$), we have,

```
Instance Sp_Comp_T_Embedding ...  '(CRC: @CartRestrictionCat ...)  (A: CRC)
( T: TuringCat ...  A )
(emb_col:  forall x:  T, { mrx:  prod (Hom x A) ( Hom A x ) |
     ((snd mrx) o (fst mrx)) = id x } ):
Embedding T (SplitCompA_justCat_allIdem ...  CRC A) .
```

Here, again, we see that the logical framework of the formal language implemented by the Coq proof assistant system is powerful enough to express the majority of the concepts, structures and proofs we encounter when studying the Turing category abstract computation model and represent its internal language.

5.6 Formalizing Range Structure

The idea of computing the range of a map is another interesting topic studied as part of the theory of recursion. The reason we have selected this specific categorical structure to formalize is, in part, the difficulty of reconciling formal and informal definitions and results about partial set maps. We have presented, in Chapter 3, a way to get away from this set-theoretic view of the range of a function so as to be able to study it in the more abstract context of a restriction category. Recall that the concept of a range of a map can be expressed in terms of idempotents — in a way, a strategy dual to expressing domains as idempotents (the restriction combinator approach). It is dual in the sense that in the opposite category of a given Turing category with range structure, the restriction combinator would play the role of the range combinator and vice-versa (see discussion in Section 3.1). This duality between ranges and restrictions gives us a new perspective on partiality that could prove especially interesting in future studies of partiality as it relates to the concept of map equality in a category (see [47], Chapter 3).

In this section, we will first describe the formalization of the range combinator definition (as originally given in [13]), then move on to describing formalizing the theorems that relate range combinators and Turing structure. The code for formalisms described here is found in the file **Range.v**.

5.6.1 Range Structure in Restriction Categories

We begin by doing a formal study of open maps in a restriction category — a notion closely related to the existence of a range combinator in the given category. Recall that an open map is an idempotent in a restriction category such that it is equal to its own restriction (see Definition 2.1.8 and the preceding discussion). The upcoming definitions formalize these concepts. Formally, we define the collection of maps on an object a (that are taken to be open) as a dependent type. For this and subsequent

definitions, see the code in **Restriction.v**.

```
Definition Op `{RC: RestrictionCat} (a: RC):= { e: Hom a a | rc a a e = e}.
```

Next, we define the combinator $(-)^*$, which takes an open map on an object a to an open map on b along a given map $f : a \to b$. It has the header

```
Definition f_star `{RC: RestrictionCat} (a b: RC) (f: Hom a b): (Op b) -> (Op a).
```

We also require a formal definition of the notion of meets of restriction idempotents in a (restriction) category, which is simply the composition of two restriction idempotents (see discussion above Definition 2.1.8). This operation is called `op_wedge` in the code. Recall that the openness of a map is characterized by the existence of the $\exists_f$ combinator satisfying certain properties (see, again, Definition 2.1.8). We next define a predicate which is true whenever a given map is open with a specific $\exists_f$ combinator *as a witness*, which the predicate accepts as a parameter:

```
open_exist_f `{RC: RestrictionCat} (a b: RC) (f: Hom a b)
(exist_f: (Op a) -> (Op b)): Prop.
```

Finally, we define a formalization of openness, which, given objects `a, b : RC` and a map `f : Hom a b`, is expressed in terms of existence of `exist_f` satisfying `open_exist_f a b f`. In the code it is called `open`. We have also formalized what it means for a map to be the partial inverse of another map — it is the predicate `partial_inverse` (see Definition 3.1.2) in our code, as well as the formulation of the lemma that states that partial inverses are always open, `crc_open` (in this thesis, Lemma 2.1.9).

The upcoming definitions have to do specifically with the range combinator concept, and their informal counterparts are specified in Chapter 3. Recall that, similar to the restriction combinator definition, the range combinator definition (Definition 3.1.1) requires the combinator $\hat{(-)}$ to send a map $f : A \to B$ to a map $\hat{f} : B \to B$ such that the axioms **RR.1, ..., RR.4** as in Definition 3.1.1 are satisfied. Formally, we define the header of the (just the mapping given by) the combinator:

```
Definition rrcType '(RC: RestrictionCat):=

forall a b:  C, Hom a b -> Hom b b .
```

Then, we define a type class that formalizes both the mapping and the behavior of the range combinator, including a term of type `rrcType RC` and proof obligations ensuring that **RR.1, ..., RR.4** are satisfied:

```
Class RangeComb '{RC: RestrictionCat}:  Type:=

{

     rrc:  rrcType RC ;

     rrc1 :  forall (a b:  Obj), forall (f:  Hom a b),

     rc b b (rrc a b f) = rrc _ _ f ;

     rrc2 :  ...  ;

     rrc3 :  ...  ;

     rrc4 :  ...  ;

}.
```

Additionally, we define a predicate `rrc5` on the type `RangeComb RC` that is true when **RR.5** is satisfied. The predicate `rrc5` need not necessarily be satisfied for all range combinators (see discussion after the 3.1.1 definition). Of course, we must also formalize the type class of categories with range structure:

```
Class RangeCat ...  '(RC: @RestrictionCat ...  )'(rrc:  @RangeComb RC ...):

Type:= { RCat_RRC: RangeComb:= rrc }.
```

Note that the `:=` symbol inside the curly braces containing the body of the type class definition is used to define the term on the left to be equal to the term on the right; and in this case, `RCat_RRC` is defined to be equal to the `rrc` parameter of the type class.

Also, just like every time we build a type class that contains a category plus a specific type of structure found in that category, we must define a coercion that allows the newly defined type class to behave as the category. To tie together the notion on

an open map and a range combinator, the following definition formulates the $\exists_f$ map in a given category with a range combinator (only the header is given here, see the code for full definition):

```
Definition exist_f `{RC: RestrictionCat} (rrc:  RangeComb )
(a b:  RC) (f:  Hom a b):  Op a → Op b.
```

For a given embedding-retraction pair in a range category, there are a number of things we can conclude about the ranges of the m and r maps, as well as the nature of the idempotent. These results are due to the author [47], and are found in Lemma 3.2.1 of this thesis (the informal version). In our code, this lemma is formalized by the `Lemma` we call **`ranges_retractions`**.

We do not give the full code for the proof of the lemma. The proof of this lemma are due to the author. Formalizing and proving them was a reassuring testament to their correctness. In particular, part (iv) for the lemma (in our code) was an interesting implication (but with an accessible formal and informal proof). The idea of this part of the lemma is that with the additional RR5 range axiom ($fh = gf \Rightarrow f\hat{h} = g\hat{h}$), a given idempotent in a range category has the same splitting as a restriction idempotent, so it suffices to considering only the restriction idempotents and their splitting in a given category for the purposes of studying its structure. The formal proof essentially performs the same sequence of rewrites as is done informally, confirming the correctness of their application.

Within the same (informal version of the) lemma, we have formalized part (ii) separately, as it does not require a given category to be a range category. It is a result about the openness of embedding and retraction maps in an arbitrary restriction category. Informally, it says that given a restriction idempotent e on A, we can express the range of r and re as follows: $r = r\hat{m}$, and $\widehat{re} = rem$. Of course, formally, we are not studying these equalities in a range category, so we must use $\exists_f$ (which defines the range combinator in a category where it exists, see discussion at the beginning of

Section 3.1 and Definition 2.1.8) instead of a range combinator.

```
Lemma ranges_retr_open `{RC: RestrictionCat}:  forall ( x y:  RC),
forall (m:  Hom x y), forall (r:  Hom y x), (r ∘ m = id x) →
(open y x r) ∧ (forall (e:  Op y) (exist_f_mr:  (Op y) -> (Op y))
(exist_f_r:  (Op y) → (Op x)), (open_exist_f _ _ r exist_f_r) →
(open_exist_f _ _ (m ∘ r) exist_f_mr) →
(proj1_sig (exist_f_r e) = proj1_sig (f_star _ _ m (exist_f_mr e))) ) .
```

As in the case of a cartesian restriction category, in order to derive any meaningful results regarding the range combinator, we must demand that range structure behave in a predictable way with respect to the cartesian structure. This predictable behavior is described by the Beck-Chevalley condition (see Definition 3.3.1). It is the predicate `Beck_Chevalley` in the code — a proposition defined for the product of any two maps. The proposition `sat_Beck_Chevalley` expresses that the Beck-Chevalley condition is satisfied for the entire collection of products of all maps in the category. Additionally, we have defined a predicate which is equivalent to the Beck-Chevalley condition, but expressed in terms of open maps, `Beck_Chevalley_open` . In the absence of a range combinator, equipped only with the openness (existence of $\exists_f$) for certain maps in the category, we must work with this version of the condition. This is a direct reformulation of the original Beck-Chevalley condition.

Recall that a range category satisfying the Beck-Chevalley condition is referred to as a cartesian range category, see Definition 3.3.1. With the range structure and the above predicates formalized, it is now possible to define a type class representing the notion of a cartesian range category:

```
Class CartRangeCat ...  `(CRC: @CartRestrictionCat ...)
(rrco:  @RangeComb ...  CRC) `(RRC: @RangeCat ...  CRC rrco ):  Type:=
{ RCat_BCC: sat_Beck_Chevalley ...  CRC rrco RRC }.
```

5.6.2 Range Structure in Turing Categories

One of the main contributions made by the author to the theory of Turing categories [47] was the characterization of range structure in terms of combinators (that is, total point maps $1 \to A$ in a Turing category with Turing object A). Recall (from Proposition 3.4.1) that when the range of a map f in a Turing category is defined, like every map in a Turing category, the map $\hat{f}$ necessarily factors through the $\bullet$ morphism. Because of this property, we can characterize range structure in a Turing category in terms of combinators.

The first approach is a direct translation of range structure into the language of combinators (see Definition 3.4.2). In a Turing category, for a map $f : a \to b$, there exists a combinator c_f through which the map f factors, $f = \bullet\langle c_f, m_a\rangle$. The range combinator c_{rf}, when applied to the c_f combinator, maps to yet another combinator $c_{\hat{f}}$. This is the combinator through which $\hat{f}$ factors via $\bullet$. This type of range combinator is a *weak range combinator* because it is dependent on the map f. In the code, we define a predicate which is true whenever, given a Turing category, a weak range combinator `rf : @point ... A` exists for the given map `f`, and call it `weak_ranComb`.

Whenever there exists a combinator c_r which behaves as a range combinator for all maps f (i.e. computes the range of every map) in a Turing category, it is known as a *strong range combinator*. In our code, we define the predicate `strong_ranComb`, which, given a Turing category, is provable whenever there exists a combinator which computes the range for all maps `f` in the category.

In the upcoming definition, we will be using the existence of a combinator with certain properties to define range structure in a Turing category. Because of this, we should not include an existing range combinator as one of the parameters when defining the range structure-characterizing proposition for a given Turing category. Therefore, our formal result is expressed strictly in terms of openness of certain maps

— in particular, of point maps $1 \to A$ and the $\bullet$ map, in a given Turing category. The `bullet_points_open_range` predicate corresponds to the main result in the Chapter 3, Proposition 3.4.1.

The parameters this predicate accepts include a cartesian restriction category, an object `A` in this category, and a formalized Turing category `T` wherein `A` is the Turing object. It also accepts a `bullet` map (a Turing morphism in `T`) as well as a term `index_col` representing the (explicit) collection of indexes for all maps in the Turing category with respect to the `bullet` map (see Definitions 2.3.1 and 2.3.2). Now, the propositional premises of Proposition 3.4.1, as well as our formalization thereof, `bullet_points_open_range`, are:

(i) the `bullet` map is open

(ii) for every embedding-retraction pair `m, r` (for every object `X`), `r ∘ m` is a restriction idempotent

(iii) all point maps `p: @point ... A` are open

(iv) for the product of any two maps in the category, the Beck-Chevalley condition is satisfied

Note that while neither the Turing morphism nor the index of a map for the given Turing morphism are necessarily unique (see discussion after Definition 2.3.1), we must make a Turing morphism and a corresponding collection of indexes explicit parameters of the lemma, because the openness of *specific* Turing morphism (rather than every Turing morphism) is a premise of the lemma.

Whenever the above premises are true, it follows that every map `f` in the given Turing category is open, that is, `@open ... f`, and it is therefore a (cartesian) range category. Note that in a range category, (ii) would be implied by **RR.5**. However, formally, we must express this condition in terms of idempotents and their

restrictions. Informally, **RR.5** is formulated using range combinator notation. Now, because we use premises (i)-(iii) to prove that a given category is a range category, the `bullet_points_open_range` result would be redundant if we ask that the given category is already endowed with range structure. Therefore, we must express an equivalent condition in terms of open maps instead.

There are more results regarding ranges in Turing categories to be formalized. However, the above formalization of the conditions under which a given Turing category has range structure, as well as the formal proof of the lemma regarding ranges and idempotents (Lemma 3.2.1), make for a sufficiently versatile and descriptive selection. It helps us to gain an understanding of the way the CIC formal language can be used to reason about ranges in an abstract computational setting. This selection also gives us a number of structures and results to instantiate with a specific Turing (range) category, Comp($\mathbb{N}$), and be able to observe that some well-known computability results formally conform to a more general model. Formalizing this example, as well as examples of all the categorical structure formalized in this chapter, will be the subject of the upcoming chapter.

Here we will point out that although we chose to focus more on ranges in Turing categories, formally studying the interaction of other structures with Turing structure could be fruitful and interesting. For example, informally, adding meets and joins into a Turing category makes it possible to state and prove the abstract versions of classical recursion results such as Rice's theorem and the undecidability of the halting problem [12]. This is a desirable direction for future development of this formalization project as it promises a way to reason about features of traditional computation abstractly, without expressing these features in terms of elements and extensional equality. The potential benefit of this abstract expression over using elements will become apparent as we discuss issues that came up and design decisions we had to make in building a formalization of traditional set-based computation in the next chapter.

Chapter 6

Formalizing Categorical Examples

In a typed formal language, and therefore in Coq, the correctness of a proof is immediately verified as soon as it is accepted by Coq as having the required type (this type is the proposition which it proves). Also, once Coq accepts a definition, it means it has been type-checked and deemed to be of the correct type. However, there is no way to verify that a definition or a proposition matches the informal definition given in category theory, and it is often not immediately obvious that it does (or does not). Instantiating type classes with specific categories and proving things about them helps to establish whether the formal definitions and propositions constructed in the process of formalizing abstract computability are in agreement with the concepts in the informally developed theory. For each concept formalized, at least one example must be defined to explore the usability and correctness of the theoretical definitions.

We have formalized the following categories as examples of each of the types of categories defined in the previous chapter (ordered from least structure to most structure). Here, we denote the full subcategory of Par of objects of the form $\mathbb{N}^n$ by $\mathrm{Comp}(\mathbb{N})_{full}$:

(i) $\mathrm{Comp}(\mathbb{N})_{full} \subseteq \mathsf{Par}$ (Categories)

(ii) Comp($\mathbb{N}$) $_{full} \subseteq$ Par (Restriction Categories)

(iii) Comp($\mathbb{N}$)$_{full} \subseteq$ Par (Cartesian Restriction Categories)

as well as the following restriction category with range structure

(vi) Par (Range Category)

(v) Par (Cartesian Range Category)

and additionally, the (cartesian restriction) Turing category

(vi) Comp($\mathbb{N}$)

6.1 Restriction Structure Instance

We begin by giving a formalized example of a restriction category. The category chosen for this example is Par, the category of sets and partial maps. The `Category` type class must be instantiated by defining `Obj, Hom, id` and `compose` (see Section 3.3). It also requires the completion of a number of proof obligations regarding composition with identity and associativity, which correspond to the equations of the informal definition of a category. We give the definitions of `obj` and `hom` below, used to instantiate `Obj` and `Hom` in the `Category` type class. The definitions of `id` and `compose`, given by terms `Id` and `Compose` below, as well as the proof obligations, are omitted (for details, see the code file "**Par_Cat.v**").

```
Definition obj:= Set.

Definition hom:= fun (a b:  obj) ⇒

{ P: a → Prop & ( forall x:  a, P x → b ) }.
```

Given two objects `a b: obj`, a term of type `hom a b` is a dependent pair consisting of a predicate `P : a → Prop` (which we refer to as the domain predicate) and a term of type `forall x: a, P x → b` (which we refer to as the mapping).

The domain predicate is provable at an `x: a` whenever, informally, `x` is in the domain of the given map. The mapping term takes two arguments: a term `x: a` and a proof of the proposition `pf : P x` (i.e. that the mapping is defined at the given `x`), and evaluates at `x` and `pf` to a specific `y` such that $x \mapsto y$ for the corresponding informal partial map.

```
Instance Par_Cat: Category :=
{
    Obj:= obj;
    Hom:= hom;
    compose:= Compose;
    id:= Id
}.
```

To be considered a restriction category, Par requires a restriction combinator to be defined, which we call

```
Definition rc_ParMap : forall a b: Par_Cat, Hom a b → Hom a a.
```

in our code. For objects `a b: Par_Cat` and map `f : Hom a b` we have defined the domain predicate `P: a → Prop` of the restriction of `f` (i.e. the first projection `P : a → Prop` of the dependent pair `rc_ParMap a b f: Hom a a`) to be equal to the domain of the predicate of `f`. It follows immediately that, given an `x : a`, this `x` is in the domain of `f` (i.e. we have a proof `P_x : P x`) whenever `x` is in the domain of the restriction of `f` (i.e. we also have a proof `P_x': P x`) since by our definition, `P' = P`. We define the second term of the dependent pair `rc_ParMap a b f` to be the (partial) identity on the first argument,

```
fun (x:a) (P_x' : P' x) ⇒ x
```

Above, we only defined a term that takes a map of type `Hom a b` and returns one of type `Hom a a`. In order to build an instance `rc_Par: RestrictionComb Par_Cat`, we use `rc_ParMap` to define `rc` in this instance of the `RestrictionComb`

type class, then complete the proof obligations `rc1, ..., rc4`. Next, we can use the category `Par_Cat` and the restriction combinator we defined in this category, `rc_Par`, to instantiate the restriction category of sets and partial maps. These two arguments are all we need for building an instance of the type class `RestrictionCat`.

```
Instance Par_isRC: RestrictionCat Par_Cat rc_Par:= { }.
```

Now, the type class definition of `RestrictionComb` requires the proof obligations corresponding to those in Definition 2.1.2 to hold, so we must formally complete them when instantiating this type class. These proof obligations are of the four defining properties of the behavior of the restriction combinator with regards to maps in the given category. This will require performing some equality rewrites. This means that some decisions have to be made about which categorical and equality structure are best suited for the task.

While the `Category` type class does not require a new definition of equality between maps to be provided, in the case of Par, it is strategic to build one into the file in the form of an axiom stating exactly when equality holds. This is done in order to facilitate proving many results about the category. The following is a definition of a predicate used in place of (and along with) the equality predicate for comparing two partial maps between a pair of objects. The idea is to define a version of functional extensionality that takes into account membership in the domain of the corresponding functions (recall the equality discussion in Section 4.3). The predicate defined below is the formalization of the notion of Kleene equality (see Definition 2.4.4).

```
Definition HomParEqv:  forall a b:  obj, (hom a b) → (hom a b) → Prop:=
fun (a b:  obj) (f g:  { P: a → Prop & forall x:  a, P x → b }) ⇒
( let (Pf, f_map):= f in
let (Pg, g_map):= g in (forall z:  a, Pf z ↔ Pg z) ∧
(forall (z:  a) (pf:  Pf z) (pf1:  Pg z), Pf z →      f_map z pf = g_map z pf1) )
```

Recall (from the sigma types discussion in Section 4.3) that the curly braces used above,

which are suggestive of set notation, denote a sigma type in Coq. Note that this definition uses `hom` and `obj` rather than the terms `Hom` and `Obj`, which we defined by `Hom := hom` and `Obj := obj` when discussing the instantiation of the `Category` type class. We use the `HomParEqv` predicate and the related axiom regarding formalizing Kleene equality (discussed below) in order to complete the proof obligations `compose` and `id` in `Category`, as well as to complete the proof obligations `r1, ..., r4` in the `RestrictionComb Par_Cat` type class. We chose to discuss the structure of the terms needed to build these classes before the equality discussion to give the reader a better understanding of how we use `HomParEqv` in completing the proof obligations.

A term `f : hom a b` is a pair of a domain predicate `P : a → Prop` and a map of type `forall z:a, P z → b` (as in the `hom` definition above). In the `HomParEqv` definition, given `f g : hom a b`, we first verify that for all `z : a`, the domain predicate of `f` at `z` is true if and only if the domain predicate of `g` at `z` is. We must next verify that for a given `z` and proofs `pf, pf1` of domain predicates of `f` and `g` (respectively) evaluated at `z`, the second terms of the (dependent pairs) `f` and `g` evaluate to the same value for both `f` and `g` at this `z` and the corresponding proofs `pf` and `pf1`. If both of these conditions are satisfied, `HomParEqv a b f g` is true.

When comparing partial maps, the informal '=' sign is almost always assumed to refer to Kleene equality. However, it is not provable in Coq (using existing equality properties or available axioms) that Kleene equality is equivalent to Leibnitz equality. This is the reason we have chosen to add the following axiom instead of defining equality in terms of setoids and setoid rewrites. That is, instead of proving that this predicate holds for a pair of maps if and only if they are (Leibnitz-, or '='-) equal, we have stated this in the formalization code as an axiom.

```
Axiom par_eqv_def: forall a b: obj, forall f g: hom a b,
```

```
HomParEqv a b f g → f = g.
```

Recall (from the partiality discussion in Section 2.1) also that partiality in the informal sense does not require comparison of proofs of an element's membership in a domain. In the formal case, in order to use the `par_eqv_def` axiom effectively, we require proof irrelevance. The proof irrelevance axiom in Coq states that given a proposition `p: Prop` all proofs `pf: p` of this proposition are equal. In particular, proof irrelevance is necessary when comparing elements as in `f x pf = f x pf1`, with `f: forall (x:a), P x → b`.

Thus, proof irrelevance is required when proving `HomParEqv a b f g`. However, even in combination with functional extensionality, proof irrelevance is not enough to be able to build a proof of the `par_eqv_def` axiom. One might consider, instead, another version of the extensionality axiom that is built into Coq,

```
Axiom functional_extensionality_dep : forall {a} {Pmap : a → Type},
    forall (f g : forall x : a, Pmap x),
    (forall x, f x = g x) → f = g.
```

It is not possible, however, to use this to prove `par_eqv_def` either. We can use the above axiom to prove Kleene equality of two maps `f g : hom a b` whenever they have the same domain predicate `P : a → Prop`, but we want to identify these maps as equal whenever they have the same domain in the informal sense. That is, if `P, P' : a → Prop` are domain predicates of `f, g` respectively, we want to be able to make an equality judgment `f = g` not only if `P = P'`, but whenever these predicates represent the same domain (i.e. for all `x : a`, we can prove that `P x ↔ P' x`). Since no existing axiom in Coq can be used to express this type of equality (Kleene equality), we have added `par_eqv_def`. It is close to being provable from `functional_extensionality_dep` directly, however, we would also need to identify any two propositions whenever one is true if and only if the other is, i.e. the following axiom would additionally need to be introduced:

```
Axiom id_prop :  forall p q, (p ↔ q) → p = q.
```

Introducing this axiom does not appear to cause an inconsistency in the language, nor to force the language to obey classical logic rules; although it may not necessarily result in a language identical to CIC. This axiom of propositional extensionality is used in related logical systems, like intuitionistic higher order logic in [34]. Of course the axiom is also true in the standard classical interpretation, in which Prop is the 2-element Boolean algebra. Such extensionality principles are also being studied thoroughly using a newly developed theory of mathematics, homotopy type theory [44].

Note that clearly, if two maps are equal, they are equal in the Kleene (`HomParEqv`) sense, so that the other direction of implication is immediate. It does not appear that the addition of this axiom will compromise the consistency of the underlying logical framework. However, this too requires more in-depth investigation. In this formalization, this axiom is used strictly in proving results about the Par category which require Kleene equality of maps in order to be provable informally. That is to say, we perform a Kleene equality rewrite (via this axiom) only if the current goal is of the form `f = g`, so the use of this axiom is strictly limited to the same cases in which it is used in an informal proof. This application of the `par_eqv_def` axiom is, therefore, limited to the same cases where Kleene equality quotient setoid rewriting would be used. However, it may be the case that the truly safe definition of this axiom must include specific restrictions on the use of it in rewriting. It is worth pointing out here that this issue of Kleene equivalence versus Leibnitz equality could be solved by applying Homotopy type theory reasoning [44], and would make for an interesting topic of future research.

It is also important to note that, despite having to choose between the two unappealing options of defining a setoid rewriting system (with multiple additional congruence relations to define and prove before it becomes sufficiently useful for the purposes of this thesis) and adding an equality-rewriting axiom (that may have consequences that may need to be guarded against), we have nevertheless made the

decision to formalize partial functions specifically as close as possible to the informal partial map definition in order to remain loyal to the intrinsically well-defined nature of a set map f in the category of sets and partial maps. That is, for all x, there is a *unique* (if it exists) y such that $f(x) = y$, which is a condition that must be explicitly enforced if $f : A \to B$ is not a (partial) function, but rather merely a relation.

The alternative of defining partial maps as a category of sets and relations did not seem as intuitive and straightforward a representation of the concepts laid out in the theory of restriction categories. In fact, the concept of a domain is built directly into the formal definition of a partial map type [4]. Furthermore, this approach is also consistent with the informal definitions of maps in cartesian restriction categories, Turing categories, (cartesian) range categories, etc.

This partial map approach is appropriate for representing the concepts mentioned above. However, as we will see later in this chapter, one is forced to resort to a *relation*-based approach for contending with partial recursive maps. The final result, however, will remain in the form of partial maps.

With the equality formalism decided on, it is now possible to instantiate Par as a cartesian restriction category by defining restriction products and restriction terminal objects, then completing the corresponding proof obligations. For example, the header for the definition of products (showing its type, but omitting the body) is:

```
Definition par_p_prod (a b:  Par_isRC): Par_isRC .
```

Now that we have define Par as a restriction category, we may now formally study its total subcategory. Informally, this is exactly the category of sets and (total) maps, Set. This category is formalized by Set category type class instance defined in the original library, `Set_Cat`. However, formally we must show that there is functor from Set to Tot(Par), and one in the opposite direction (for the code, see **Par_Cat.v**), declared as follows

```
Definition TotPar_Set_Cat_Eqv_f:
```

```
Functor Set_Cat (Tot rc_Par Par_isRC).

Definition TotPar_Set_Cat_Eqv_b:

Functor (Tot rc_Par Par_isRC) Set_Cat .
```

such that these are defined on maps and objects of the categories, and we can prove that these are indeed functors (the necessary equalities hold). Furthermore, we require that these two functors are indeed inverses of each other, on both objects and maps of the categories. To do this we have proved four lemmas, one showing that the composition of the above functors gives the identity on the objects of Tot(Par), a similar result about applying the composition of the two functors to maps in this category, and two more about maps and objects of Set. These lemmas demonstrate that the above functors compose both ways to give the identity functor. The following code formalizes the first of the lemmas (about objects of Tot(Par)):

```
Lemma TotPar_Set_Cat_Eqv_o:  forall a ,

@FO (Tot rc_Par Par_isRC) (Tot rc_Par Par_isRC)

(Functor_compose TotPar_Set_Cat_Eqv_b TotPar_Set_Cat_Eqv_f) a =

@FO (Tot rc_Par Par_isRC) (Tot rc_Par Par_isRC) (Functor_id _) a.
```

The term `FO` above is a term inside the `Functor` type class which is a mapping between the object of the given categories. The term `FO` takes three implicit parameters, which consist of two categories and an instance of `Functor` type class representing the functor between them (to which `FO` belongs). Above, both objects are `Tot rc_Par Par_isRC` in both cases `FO` is used. On the left of the equality, the functor `F` (the third implicit argument of `FO`) is the composition of functors `TotPar_Set_Cat_Eqv_f` and `TotPar_Set_Cat_Eqv_b` (with composition formally defined in the existing category theory library class **Functor.v** by `Functor_compose`), and on the right, this is the identity functor `Functor_id`, also defined in the library.

The (formal) provability of this result is a very good indicator that Par is indeed a bigger category than Set, containing a subcategory isomorphic to it. It is straight-

forward to formally define maps which are not total and do not belong in Set. To clarify this important point, recall the category of pointed sets [21], denoted $\mathsf{Set}_\perp$. We can associate (via an essentially surjective functor), for each map $f : A \to B$ in Par, a map $f_\perp : A_\perp \to B_\perp$ in $\mathsf{Set}_\perp$ which evaluates to $\perp$ at all the points x on which f is undefined. These categories are essentially the same informally. If we were to try to formalize $\mathsf{Set}_\perp$, we would see that we are forced to do so in a manner very similar to our Par formalization. We would again require a predicate which determines, for each $x : A_\perp$, whether this x should be mapped to $\perp$ or some value $y : B_\perp$ by the given $f_\perp$. And once again, if $f_\perp(x)$ exists, we would require a proof of this. If such a proof does not exist, and we are not working in the context of classical logic, we cannot prove that this predicate evaluates to false. For this categorical formulation of the completion of partial functions, there is still no way for a map $f : A \to B$ in the $\mathsf{Set}_\perp$ category to formally evaluate to $\perp$ whenever there is no proof that $f(x) \downarrow \Rightarrow \perp$, e.g. when we cannot prove that a computation will never terminate.

We have also instantiated Par as a cartesian restriction category as follows:

```
Instance Par_isCRC : CartRestrictionCat rc_Par .

exists.  exact PPTerm.  exact PPProds.
```

Here,

```
PPTerm :  @Has_pProducts Par_isRC rc_Par Par_isRC.

PPProds :  @ParTerm Par_isRC rc_Par Par_isRC.
```

are terms representing the formal products and terminal object in `Par_isRC`, which we have also defined in our code. They are formalized according to the notion of products in Par as in Example 2.2.3. The tactic `exists` instantiates a type class whenever it does not require additional terms for instantiation.

Next, we expand our notion of the category Par to include range categorical structure. Recall that Par has been informally shown to admit a range combinator, which exactly corresponds with the classical notion of the range of a function (see

discussion in Section 3.1). We now proceed to build the formal definition range structure in `Par_isCRC`. Note that the reason we require the full cartesian restriction category structure here is that it will be easier to later prove that it admits not only range combinators, but cartesian range structure.

Just as in the case of the restriction combinator, we must first give the definition of the actual combinator (the term that gives the $\hat{f}$ map for a given f, informally). Recall from the discussion in Section 5.6.1 that this means defining a term, which we call `rrc`, whose type is given by applying the term `rrcType`, with the following header

```
rrcType '(RC : RestrictionCat) := forall a b, Hom a b → Hom b b
```

to the cartesian restriction category `Par_isCRC`. The term `rrc` is defined to correspond closely to the informal setting (as in the motivating example in Section 3.1). For a map `f : Hom a b` (a dependent type pair) in `Par_isCRC`, we can define `rrc a b f` in terms of the domain predicate `P : a → Prop` of `f`. We define the first of the two terms which make up the dependent pair `rrc a b f : Hom b b`, that is, its domain predicate, as follows

```
P' := exists (fun (y: b) => (exists x: a, exists p: (P x), f x p = y))
```

Then, we define the second term of the dependent pair `rrc a b f`, which has the type `forall y:b, P' y → b`, as a (partial) identity on the first argument. This map evaluates to `y:b` at `y, pf`, for any proof `pf` of `P' y`.

We may now use this `rrc` definition to instantiate the `RangeComb` type class, the type class that formally represents the range combinator (see Section 5.6.1). We also complete the required proof obligations [RR.1, ..., RR.4], as in definition 3.1.1. With this range combinator type class instance (which we call `rrcPar`) and the `Par_isCRC` category, we build the instance `Par_isRangeC` of the `RangeCat` type class corresponding to the informal definition of Par with range structure.

However, we need one more ingredient, a proof that the range combinator satis-

fies the Beck-Chevalley condition, see Definition 3.3.1. Recall that this condition is satisfied whenever given two maps f, g in a cartesian restriction category, the range of $\langle f, g\rangle$ is equal to $\langle \hat{f}, \hat{g}\rangle$. This is straightforward to prove in a category of sets and partial maps, where a product of two maps, $f \times g : A \times B \to C \times D$, is defined by $(x, y) \mapsto (f(x), g(y))$. This result in our code is formalized by the lemma `Par_BCC`. Finally, we have all the definitions and proofs (which fulfill the necessary proof obligations) to instantiate the cartesian range category of sets and partial maps:

```
Instance Par_isCRRC: CartRangeCat ...  Par_isRangeC.

exists.  exact Par_BCC.
```

This is the extent to which the category of all sets and (all) partial maps between them can serve as an example of structure found in Turing categories. Par does not contain a morphism that computes every map in the category, so it does not admit Turing structure. One justification of this is that by Lemma 2.3.3, every object in a Turing category is a retract of the Turing object. There cannot exist such an object in Par since for all A : Par, there is no injective map of the power set of A into A, by Cantor's Theorem [26], page 110.

6.2 Formalizing Traditional Computability as a Turing Category

Next, we move on to formalizing the motivating example of Turing category theory. This motivating example, the category Comp($\mathbb{N}$), must be considered as a subcategory of Par, as it is made up of a subcollection of objects of Par (the n-fold products of the set of natural numbers), and a subcollection of maps between these objects (only those that are Kleene-equal to partial recursive maps). The work described in this section is incomplete, but relevant to the topic studied. We have included it in this thesis because a significant portion is done, and the rest should be straightforward

but time consuming. The code for the formalization described in the rest of this chapter is found in our **CompN_Cat.v** file.

The formalization of partial recursive maps (prf's) in this category follows an existing Coq encoding, with the specifics of the enumeration of prf's and the proofs of the prf $\leftrightarrow \mathbb{N}$ bijection resulting from this enumeration completed in [50]. For this reason, and because the focus of this formalization is categorical modeling of computation and formally formulating the results of traditional computation in a way that conforms to this abstract model, we have omitted many of the computational proofs about the nature of the computation in this bijection.

Recall that every partial recursive (also referred to computable, or simply recursive) map ('prf'), is constructed from the following set of functions and rules [20]:

Definition 6.2.1 The partial recursive functions are the smallest class $\mathcal{C}$ of partial numerical functions containing the following functions

(i) The successor function $S : \mathbb{N} \to \mathbb{N}$ defined by $S(n) = n + 1$.

(ii) The zero function (or constant function) $Z : \mathbb{N} \to \mathbb{N}$, defined by $Z(n) = 0$.

(iii) Projection functions $P_n^i : \mathbb{N}^n \to \mathbb{N}$, defined by $P_n^i(x_1, \ldots, x_n) = x_i$.

and closed under the following rules:

(iv) Composition: if $g_1, ..., g_m : \mathbb{N}^k \to \mathbb{N}$ are in $\mathcal{C}$, and $f : \mathbb{N}^m \to \mathbb{N}$ is in $\mathcal{C}$, then the composite function $f(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k)) : \mathbb{N}^k \to \mathbb{N}$ is in $\mathcal{C}$.

(v) Primitive Recursion: If $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ are in $\mathcal{C}$, then so is $f : \mathbb{N}^{k+1} \to \mathbb{N}$, where $f(0, x_1, ..., x_k) = g(x_1, ..., x_k)$ and $f(S(y), x_1, ..., x_k) = h(y, f(y, x_1, ..., x_k), x_1, ..., x_k)$.

(vi) Minimalization: If $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is in $\mathcal{C}$, so is $g : \mathbb{N}^k \to \mathbb{N}$ defined as follows: $g(x_1, ..., x_k) = c$ if and only if $f(c, x_1, ..x_k) = 0$ and for all $0 \leq d \leq c - 1$,

$f(d, x_1, ..x_k)$ is defined and $f(d, x_1, ..x_k) > 0$.
We write $g(x_1, ..., x_k) = \mu c.(f(c, x_1, ..., x_k) = 0)$ to mean the least c such that $f(c, x_1, ..., x_k) = 0$, where for all smaller values d, $f(d, x_1, ..., x_k)$ is defined and not equal to 0.

In the attempt to formalize the above definition, we must consider the following observations:

(i) A prf f defined in terms of the above six functions need not halt on every input (so, its domain is not all of $\mathbb{N}$).

(ii) We have no way of defining the domain of f other than in terms of f itself, i.e. x is in the domain of f when there exists a number y such that $f(x) = y$.

(iii) Because of observation (ii), we cannot immediately mold a prf f into the partial map formalism used in `Par_Cat`, since in the dependent pair representing a partial map,
`{ P: A → Prop & forall x: A, P x → B }`
the partial map depends on the (domain) predicate, but never the other way around.

6.2.1 Existing Formalizations of Recursive Maps in Coq

Let us begin by considering the following formalisms pertaining to enumerating recursive functions — this appears to be the extent of existing formalization effort in this direction. The first is due to Russell O'Connor [37]. This work focuses heavily on formalizing formulas and proofs. A formalization of primitive recursive maps is also completed as an example of a system of axioms to reason about using the defined language. The approach taken in O'Connor's work is formalizing strictly primitive recursion.

The set of maps defined in terms of constructors (i) - (vi) in Definition 6.2.1, coincides with a number of other definitions of computable functions such as register machines, etc. It represents the nature of computable functions according to the Church-Turing thesis [20], and is the set of maps generally considered in the study of traditional computability. This is the collection of maps that informally form the category of computable maps that is the motivating example of a Turing category. This is the reason it is preferable to consider a formalization of partial recursive maps rather than primitive recursive ones. Furthermore, the partial recursive map category admits a non-trivial restriction combinator, yielding a much more interesting and representative example.

Moreover, the term defining the syntax of primitive recursive maps given in [37], `PrimRec: nat → Set`, is accompanied by the following term, which represents the semantics of the language:

```
evalPrimRec: forall n: nat, PrimRec n -> naryFunc n
```

This term, for a given `n : nat` and a primitive recursive function `f` expressed in the `PrimRec` language, defines a map on `n` (natural number) arguments which is equal, via functional extensionality, to the primitive recursive computation corresponding to `f`.

Recall (from our discussion of formalization of partial maps in Section 6.1) that it is impossible to use functional extensionality in its usual form to compare partial maps. So, this definition is rather inconvenient to adjust to conform to the structure of `f: Hom A B` in the `Par_isCRC` category, even if we modify it to include the full set of partial recursive maps. These differences in defining partial and primitive recursive maps are addressed elegantly in the formalizations representing the language of partial recursive functions, which we discuss next.

Next we consider the formalization of partial recursive computation given in [17] by Constable and Smith. In this work, the language of recursive maps is defined

directly in terms of numbers and lambda abstractions, rather than by using proof assistant software. The semantics of the theory are given by defining a reduction relation. To evaluate computation, the notion of machine is defined by means of a relation. This relation follows closely the set of constructors required for recursive computation in Definition 6.2.1, and is similar to the formal approach we have taken.

Yet another formalization of computable maps has been done in a different formal proof assistant, HOL [48]. The approach taken in the formalization of computation by register machines involves directly referencing an `Undef` value, the formal equivalent of $\uparrow$. This is a deterrent from taking this approach, as it is more difficult to adapt to the theory of partial maps on which Turing categories rely. For a comparative study of formalizing partial recursion in HOL and Coq, see [49].

The definition of partial recursive maps, Definition 6.2.1, is also (sufficiently directly) translated into a formal setting using our proof assistant of choice, Coq. This formalization takes into consideration the above observations (i) - (iii). The upcoming definitions are due to Vincent Zammit, as described in his paper [50], and we follow it exactly. An important factor in using this definition in the formulation of the desired category (rather than attempting to formulate a computable map category directly) is the key role the proof of the S_n^m theorem plays in the definition of Turing structure in this category.

As we observe in the `prf` formalization work of Vincent Zammit, a formal proof of the S_n^m theorem is far from trivial, requiring multiple (inevitably awkward) formal definitions of maps that are well-known to be computable. For the formalization code of the partial recursive maps and the associated category Comp($\mathbb{N}$) see **CompN_Cat.v**.

```
Inductive prf:=

    | Zero:  prf

    | Succ:  prf

    | Proj:  nat → prf
```

```
| Sub:  prf → prf → nat → nat → prf

| Rec:  prf → prf → prf

| Min:  prf → prf.
```

Note that this is the first time the keyword `Inductive` appears in this thesis. It is used in Coq for introducing inductively defined data and propositions (in place of `Definition` for non-inductive types).

The `prf` definition gives the constructors from which any partial recursive map can be built inductively. These constructors are simply the signifiers representing the basic partial recursive maps, which is to say, the term `prf` gives only the syntax of the formal language of partial recursive functions. The resulting prf's do not, themselves, have a domain and cannot be evaluated on an input — thus, avoiding the problem in observation (iii).

Next, we may give a relation (rather than a partial function) to specify the semantics of when, on an input x, a given prf f converges to a number y. The following inductively-defined relation (also due to Vincent Zammit in [50], Section 4.3) does exactly that (here we only show a small part of the full definition of `converges_to`):

```
Inductive converges_to:  prf → list nat → nat → Prop:=

    | conv_zero:  forall (l:  list nat), converges_to Zero l 0

    | conv_succ_nil:  converges_to Succ nil 1

    | conv_succ_cons:  forall (x:  nat), forall (l:  list nat),

        converges_to Succ (x::  l) (x + 1)

    | ...
```

The proposition `converges_to f lx y` is true whenever (informally) the given prf f, applied to a list of numbers $lx = (x_1, ..., x_k)$, outputs the number y. Informally, a computation is performed on a tuple of numbers. In the inductive definition above, we represent a tuple of numbers as a Coq list. The notation `x ::  l` is shorthand for `cons x l`, which is a term that adds the element `x` to the front of the list `l` (i.e.

x is the head of the new list, and l is the tail).

Here we would like to point out that it appears possible to build a language A_prf which formalizes computation with an oracle for a subset

```
Definition A := { x:  nat | P x }.
```

of the natural numbers, where P : nat -> Prop is a predicate which is true whenever, informally, $x \in A$. For details, see Example 2.4.6. In order to do this, we would need to modify the original definition of the prf language by adding a constructor Xa : prf, as well as the following constructors to the converges_to semantics:

```
| conv_Xa_T : forall n, (P n) -> converges_to Xa (n::nil) 1

| conv_Xa_F : forall n, (∼(P n)) -> converges_to Xa (n::nil) 0

| conv_Xa_or :  forall n:nat, ∼(∼(P n)) -> (converges_to Xa (n::nil) 1).
```

As required for computation with an oracle, the three constructors explicitly enforce the totality of the Xa map due to the domain being classically defined. That is, proving that Xa is defined on n can be done by demonstrating that ∼(∼(P n)) (as well as demonstrating P n directly). Now, in order to define Turing structure within this category, we would still need to formalize the enumeration of A-computable maps, prove the A-computable version of the S_n^m theorem, and prove that the universal applicative map is A-computable. With these proofs and definitions in place, the proof that Turing structure diagrams in this category commute would coincide exactly with the formal proof for classical (partial) recursive map category discussed below.

6.2.2 Defining a Category Using the prf Language

At this point, a category representing traditional computability could be defined directly, consisting of objects represented by natural numbers (i.e. $\underline{n}$ represents $\mathbb{N}^n$). That is, since prf expresses the semantics of partial recursive maps as a relation (the converges_to predicate), we could have instead built the category of partial recursive maps as a subcategory of sets and relations. In such a category, a computable map

morphism $f : \underline{n} \to \underline{m}$ corresponding to `f : prf` would be represented by the relation `converges_to f`. However, we have chosen to build this category in a different way, as a subcategory of Par.

The motivation behind the design decision we made regarding the formalization of the Comp(ℕ) category in terms of partial maps rather than relations is the desire to maintain the subcategory hierarchy imposed by the definition of Comp(𝔸) (see discussion in Section 2.4), which inherits its restriction combinator and cartesian structure from the larger category containing A (of which Comp(𝔸) is a subcategory).

Recall that we have already defined, in **CompA.v**, an instance of a cartesian restriction category of all maps (not just the computable ones) of the form $A^n \to A^m$, see Section 5.5.1. This instance takes a number of arguments, including the larger category and the object `A`, needed to build the category of powers of A. To obtain the full subcategory (of Par) of n-fold products of the natural numbers and $\mathbb{N}^n \to \mathbb{N}^m$ maps, for example, we must supply this instance with arguments as follows:

```
Definition CompNCRC:= CompA_CRCat Par_Cat rc_Par Par_isRC Par_isCRC nat .
```

The (here omitted) definition of `CompNRC` is very similar — it defines the same category without the cartesian structure, and it is a term of type `RestrictionCat ...`. We have also, similarly, using the terms discussed in Section 5.5.1, defined a term for the restriction combinator structure `rc_CompN : RestrictionComb ...` in both of the above categories, `CompNCRC` and `CompNRC`. Finally, we have also defined a term of type `Category`, without any additional structure, called `CompsNR` in our code. The `Obj` and `Hom` terms in the three instances of type classes of categorical structures, `CompsNR, CompNRC` and `CompNCRC`, are the same as the `Obj` and `Hom` terms, respectively, across all three type classes. There are coercions already defined that allow us to treat an object in one of the classes as an object in either of the other two, and a map in any of these as a map if either of the other two. These instances of type classes differ only by additional restriction or cartesian restriction structure

required to instantiate each type class.

As noted earlier, `converges_to` is not a function but rather a relation; it is not immediate that, given an `f: prf`, `converges_to f` must correspond to a (well-defined) function of type `list nat → nat`. Since it is our intent to build a partial map category, we must make use of this definition to reason about partial recursion in terms of functions. This means that we must have a proof of well-definedness of the partial map corresponding to this relation. That is, for every pair of a function `f : prf` and a list `ln: list nat`, whenever `converges_to f ln y` is provable, the number `y: nat` must be *unique*.

We call this proposition `unique_conv` in our code. Given a list `ln` of natural numbers, an `f : prf` and `y z : nat`, the proposition `unique_conv f ln y z` has two premises, which we call here `H1 : converges_to f ln y` and `H2 : converges_to f ln z`. The goal, `y = z`, is proved by induction on the hypotheses `H1` and `H2`. There are a total of twelve constructors for each of `H1` and `H2`, each of the form

```
⟨ premises ⟩ → cnst ⟨ arguments ⟩
```

where `cnst` is a constructor of the inductive type `prf`, which is either `Zero, Succ, Proj, Sub, Rec`, or `Min`. In the inductive cases where the types of the `cnst`, as well as the arguments, if any (see the `prf` definition), match for premises `H1` and `H2` (i.e. both are, for example, `Succ` — for a total of twelve non-trivial cases) we must demonstrate that indeed `y = z`. We have done so for the first five constructors of `converges_to`, leaving the rest for future work. In the cases where the arguments or `cnst` do not match for `H1` and `H2`, we have a proof of an equality such as `Zero = Succ` as a hypothesis. This is a false statement, from which any other statement can be proved in formal logic (that is, these are trivial cases).

At this point in the code, we introduce the first (type-specific) instance of the axiom of choice. We want to define partial maps and reason about a specific nat-

ural number `y` that a given partial map evaluates to on a given input. To use the `converges_to` predicate for this purpose, we will require to choose a specific `y: nat` satisfying the `converges_to f ln x y` predicate.

To make this choice, we simply define a parameter that, given an arbitrary `f: prf`, an `ln: list nat`, and a proof (`pf_ex` below) of the existence of a `y: nat` satisfying `converges_to f ln x y`, supplies us with a natural number:

```
Definition AC_select_y (ln: list nat) (f: prf)
(pf_ex: exists (y: nat), (converges_to f ln y)) : nat.
```

Now, we do not know the specific value of `AC_select_y ln f pf_ex` — being able to define it explicitly would eliminate the need for the axiom of choice altogether. However, we can demand that it satisfies the property that `converges_to f ln (AC_select_y ln f pf_ex)`, formalized as the axiom:

```
Axiom AC_rewrite: forall (ln: list nat) (f: prf)
(pf_ex: exists (y: nat), (converges_to f ln y)) ,
converges_to f ln (AC_select_y ln f pf_ex).
```

Note that even if we have a proof `pf_ex` that such a `y` exists, we cannot directly use this fact to build a fixpoint method to compute the desired value. There is no argument (or metric) that must decrease in the corresponding recursive method (more on this in Section 6.2.4). Therefore, Coq does not allow for the definition of this computation. This is why we require the `AC_rewrite` axiom, which states that `AC_select_y f ln pf` defines the natural number that the computation `f : prf` converges to on a given input (whenever we have a proof that it does indeed converge to *some* value on the given input). In the formal language underlying Coq, however, when the current goal is of type `Prop` and one of the premises is a proof of some existential statement `pf : exists x:A, P x`, it is possible to pick a specific `x` satisfying this property, i.e. add `x:A` and `pf_x: P x` to the list of premises [18]. That is, in this case, when proving a proposition with `pf_ex` as a premise, it is indeed

possible to select the required `y` without the use of the `AC_rewrite` axiom.

We now proceed with the definition of the subcategory of Par containing only the computable maps of the form $f : \mathbb{N}^n \to \mathbb{N}^m$. In order to define a wide subcategory using the library built by Timany and Jacobs [45], we must define a predicate on the maps in the larger category so that a given map is contained in the subcategory whenever the predicate applied to this map is true. Then, the maps in the resulting subcategory are pairs (sigma types) consisting of a map in the larger category, and a proof of the predicate applied to this map. In this case, such a predicate must formalize the notion of computability. That is, we define a predicate on maps in the `CompsNR` category which we take to be the formal definition of the computability of a map `f : Hom a b`, for some objects `a b : CompNCRC`. From now on, when we refer to a map as computable, we mean that the following recursively defined predicate is true:

```
Fixpoint conv_to_cat_prop (n m: nat)
(f: @Hom CompsNR (build_compsNR_obj n) (build_compsNR_obj m)): Prop:=
match m with
     | 0 => (conv_to_cat_zero_fix n n (@rc _ rc_CompN _ _ f))
     | S 0 => (exists (prf_f : prf) , prf_par_map prf_f (build_compsNR_obj n)
          = (pr1f n m f))
     | S (S m') ⇒ (conv_to_cat_one n m f conv_to_cat_prop)
end.
```

We do not explain the details of the above code, but instead, we give the reader a more high-level explanation of the proposition defined in each of the cases. In the above fixpoint definition, the term `build_compsNR_obj n` defines an object in the `CompsNR` category which informally corresponds to $\mathbb{N}^n$. The general idea of the above recursive definition is that the proposition generated for a given map `f` (informally, $f : \mathbb{N}^n \to \mathbb{N}^m$) is true when there exists some collection of m terms in `prf` such

that the i^{th} term represents a (partial recursive) map that is Kleene-equal to the map $f_i : \mathbb{N}^n \to \mathbb{N}$ in the i^{th} component of `f`. Such a collection is said to formally compute `f`. The case for $m = 0$ requires special consideration, which we will discuss later, when we give the details about each of cases in the fixpoint.

In order to build if a map $f_i : \mathbb{N}^n \to \mathbb{N}$ represented by a term of type `prf`, we define the term `prf_par_map`. To define the term `prf_par_map prf_f ...`, we need to first define the domain (the `P: (nthProdC ... nat n) → Prop` predicate) of the partial map that we are building. For a term `x : nthProdC ... nat n`, we define the corresponding proposition regarding the membership of `x` in the domain of the partial map we are defining. We do this by first converting the object `x : nthProdC ... nat n` to one of type `list nat`, which is done by applying the term `N_toProd` to the term `x`. This term (naturally) identifies the element `x = (x1, (x2, ...,xn))` with the list `(x1,...,xn)`. We write `dom_x` to stand for the term

```
dom_x:= exists (y: nat), (converges_to prf_f (N_toProd x _) y)
```

Now, given `x`, a map `prf_f : prf`, and having proof of the `dom_x` predicate for `x` (informally, proof of membership of x in the domain of f), we can define the `y: nat` that the partial recursive map `prf_f` must converge to. Since we know such a `y` exists (due to having a proof of `dom_x`) and is unique (as discussed above), we may use the axiom of choice `y`-selector term `AC_select_y` to obtain the specific `y` that the partial map converges to.

We now give the outline of the cases in the `conv_to_cat_prop` fixpoint above. Note that the recursive call takes place when term `conv_to_cat_prop` is passed the term `conv_to_cat_one` within the fixpoint. It is important to point out that the fixpoint `conv_to_cat_prop` does pattern matching on *m only*, however, the (fixpoint) `conv_to_cat_zero_fix` term inside the pattern matching clause additionally does recursion on n in the case when m=0. The following is a summary of all the resulting cases, broken down by m and n values, generated within the fixpoints.

Note that in the code, we use further auxiliary definitions which are used to generate the case (and which we do no show here). This is done in order to more conveniently express the required primitive recursive (fixpoint) term so that Coq is able to recognize it as primitive recursive (on a single variable). However, below, we present the general schema of the resulting definition in terms of both variables `n` and `m` to make the explanation of the inductive sub-cases of `conv_to_cat_prop` (some of which are specified within auxiliary predicates) easier for the reader to parse.

```
(n = 0) :              (exists prf_f :  prf,
                       projT1 (prf_par_map prf_f RCat_term) tt ↔ projT1 f tt)
(n = S n', m = 0) :    exists (prf_f:  prf) ,
                       prf_par_map prf_f (build_compsNR_obj (S n)) =
                       ((pr1 (build_compsNR_obj (S n)) n H) ∘ (@rc _ rc_Par _ _ f)) ∧
                       (test_prop (S n) m (pr2Cf (build_compsNR_obj (S n))
                       (build_compsNR_obj (S m)) f m H1) ) .
(n = S n', m = S 0) :  (exists (prf_f :  prf) ,
                       prf_par_map prf_f (build_compsNR_obj n) = (pr1f n m f))
(n = S n', m = S (S m')) : ((exists (prf_f:  prf) , prf_par_map prf_f
                       (build_compsNR_obj n) = ((pr1 (build_compsNR_obj (S m')) m' H) ∘ f) )
                       ∧ (test_prop n m' (pr2Cf (build_compsNR_obj n)
                       (build_compsNR_obj (S m')) f m' H) ))
```

Case n = 0. The first case, is applicable whenever the given map `f` is of the form

`f : Hom (build_compsNR_obj 0) (build_compsNR_obj m)`, i.e.

`f : Hom RCat_term (build_compsNR_obj m)`

as the 0-fold product of `nat` with itself is exactly the terminal object in the full subcategory (of `Par_isCRC`) of n-fold products of `nat`. Informally, there are only two

kinds of maps of this type, those that are defined on $* \in 1$ (the total point maps), and those that are undefined on $*$ (i.e. defined to be $\uparrow$). The term `tt : unit` is the sole inhabitant of the terminal object in the larger category, `Par_isCRC`. So, if `P : RCat_term → Prop` is the domain predicate of `f`, it may not be provable that either `P tt` or not `P tt` (i.e. in Coq, `P tt → False`). We formally express here the notion that the proposition `conv_to_cat one f ...` is provable for a map `f` with a terminal object origin whenever there exists a `prf_f : prf` such that it converges to some natural number on the empty list if and only if `P tt` is true (i.e., informally, the given map and this `prf_f` have the same domain). We say that such a `prf_f` computes the given `f`.

Remark : In our code, we also define explicitly the `prf_f : prf` maps that compute `f` for the cases when `m = 1` and either `P tt` or not `P tt` is provable. That is, we have shown explicitly that all total maps of the form $1 \to \mathbb{N}$ (i.e. point maps) are computable. Suppose `p : point ... (build_compsNR_obj 1)` is a point map that sends `tt` to some natural number `y : nat`. We have shown that for every `y : nat`, there is a `prf_f : prf` that computes it. This is proved by lemma `term_maps_prf` in our code. To prove this lemma, we define a fixpoint `x_out (x:nat) : prf`. For every `y`, we show that for the resulting `x_out y`, `converges_to (x_out y) ln y` holds for every list `ln`. We prove this in lemma `test_x_out`, and `points_comp` is an immediate consequence.

Moreover, we show that all non-total (non-point) maps

```
np : RCat_term → (build_compsNR_obj 1)
```

such that there is a proof of `P tt → False`, are computable by the `prf` language. Informally, there is only one such map, and it corresponds to $* \mapsto \uparrow$. That is, we must show that there exists a `np_prf` such that for any list `ln`, if it converges to `y` on `ln`, `False` is provable. We have done this in lemma `undefined_point` in our code, by defining a `prf` such that it uses `Min` and `Succ` constructors to find the minimal `y`

such that `S y` converges to 0.

The existence of more that just the above two possibilities for a partial map from the terminal object into another object in the category is a particular example of a key difference between the intuitionistic logic formalization we are building and the informal definitions of traditional recursion theory. In the formal case, we may not have either a proof that a given map halts on $*$ (its domain predicate is true at `tt`), or that it does not halt (its domain predicate is false at `tt`). When we consider, in particular, maps out of a terminal object into $\mathbb{N}$, we cannot prove that either $* \mapsto\uparrow$ or $* \mapsto y$ for some y as there is no way to perform case analysis on the structure of the domain predicate `P` of a map with terminal object origin at $*$ (such a proposition is not necessarily inductively defined and may be expressed via `prf` as computation for which there is no proof that it either terminates or that it does not). We believe, however, that the possibility that neither a proof of membership in the domain of a given map, nor a proof to the contrary exist is representative of the nature of the halting problem. Note that in the results presented in [17], the authors specifically point out that there is no term (in the theory presented in the paper) to solve the halting problem. In the partial map formalization we are using, assuming there is always a proof of `P x` or of `P x → False`, where `P` is the domain predicate of a computable map `f : hom a b`, would imply that there exists a proof of the totality of the `prf` representing the characteristic function of the halting set,

$$f(i,x) = \begin{cases} 1 \text{ if } i \cdot x \text{ halts} \\ \\ 0 \text{ otherwise} \end{cases} .$$

Thus, we may not make such an assumption.

Case (n = S n', m = S 0). Here, we build the computability proposition explicitly, without making additional recursive calls. That is, this proposition is defined to be true when there exists a `f_prf : prf` such that the corresponding map `prf_par_map prf_f (build_compsNR_obj n)` is Kleene-equal to the given `f`. We say that this `f_prf` *computes* `f`. To clarify, the predicate `conv_to_cat_prop n`

`1` evaluated at a map

```
f:  @Hom CompsNR (build_compsNR_obj n) obj_nat
```

is provable when there exists a `f_prf : prf` such that the domain of `f` is all `x: build_compsNR_obj n` such that for some y, `f` converges to the list on natural numbers of length n corresponding to the n-fold product `x` of natural numbers (informally, we would write $x = (x1, ..., xn)$). Here, it is interesting to note that the domain predicate is expressed very similarly to the predicate that would reflect the range of a given `prf` — not surprising, since the semantics of the language of partial recursive maps is given by a relation, `converges_to`, rather than a function. We will elaborate on this discussion in Section 6.2.4.

Case (n = S n', m = S (S m')). The (n = S n', m = S (S m')) case corresponds informally to maps of the form $\overline{f} : \mathbb{N}^{Sn'} \to \mathbb{N}^{S(Sm')}$. The proposition we build in this case is a conjunction of two propositions. The first is true if the first component of the map f is computable (by some `f_prf : prf`), i.e. $\pi_1 f$ is computable. The second is defined recursively, making a call to `conv_to_cat_prop` from within this fixpoint term. This call builds a proposition for the rest of the `(S m')` components. That is, this second proposition is true whenever $\pi_2 f$ is computable, if we consider `f` as a map into $A \times (A^{Sm'})$. The base case of the recursive call made is the case described above, (n = S n', m = S 0). This way, at each iteration of the recursive call, we verify (i.e. build a proposition that is true whenever we can prove) the existence of a term `f_prf` that computes the first component of the given map, and make a recursive call verifies the existence of the rest of the collection of (S m') terms of type `prf` that compute the rest of the components.

Case (n = S n', m = 0). The proposition we define for this case is true whenever a map of the form `f : Hom (build_compsNR_obj n) RCat_term` (informally, $f : \mathbb{N}^n \to 1$) is computable. Informally, according to Definition 2.4.1, a map of this form is computable (i.e. factors via the given PCA and a total map) whenever its

restriction, `rc _ _ f` (informally, $\overline{f} : \mathbb{N}^n \to \mathbb{N}^n$) is computable. We note that $\overline{f}$ is computable if and only if $\langle \pi_1 \overline{f}, ..., \pi_1 \overline{f} \rangle$ is computable. We handle this case similarly to the (n = S n', m = S (S m')) case, but for the $\overline{f}$ map instead of f. That is, we again build two propositions, for $\pi_1 f$ and $\pi_2 f$ (this one, recursively). Here, again, we verify the existence of a term computing the first component of f (i.e. $\pi_1 f$), and then recursively verify the existence of the rest of the collection of the n-1 terms computing the rest of the components. However, the base case of this fixpoint $\overline{f} : \mathbb{N}^n \to 1$ (n = S n', m = 0), is the (n = 0) case. Note that both the cases (n = S n', m = S 0) and (n = S n', m = S (S m')) cases involve a recursive call (no other cases do).

Now, the predicate `conv_to_cat_prop n m f` holds when a given partial map `f: @Hom CompsNR` can be defined in terms of an m-fold product of maps computable via the `prf` language. We express this predicate in terms of objects `a b: CompsNR` as opposed to the powers of `nat`, and call it `Comp_mapsN`. Informally, this corresponds to referring to objects A^n as opposed to simply by the corresponding power n. We take this predicate `Comp_mapsN` to be the formal definition of the informal property of being 'computable' or 'partial recursive' of a map in the `CompsNR` category. That is, when this predicate is provable for a given map, we say that this map is *computable*.

Note that in the previous paragraph, we discuss computability in terms of maps between objects in the category `CompNCRC` directly, i.e `a: CompNCRC`, versus referring to the objects in terms the natural number corresponding to the power of `nat` for the given object, i.e. `(build_compsNR_obj n)`. The reason we are able to do this is that we have defined another version of the axiom of choice, as follows.

Recall from Section 5.5.1 that we did not require the axiom of choice for defining the full (cartesian restriction) subcategory of all n-fold products of an object A of a given category C. In this section, we have used the general definition of the full cartesian restriction subcategory of n-fold products A^n (for a given an object A) to build the instance `CompsNR`, where the larger category is `Par_isCRC`, and the object A is the set of all natural numbers `nat`. In order to define new maps in this category,

however, that are not inherited from the larger category (recall the discussion in Section 5.5.1, again), we must often use the defining property of the objects in this category: they are objects of `Par_isCRC` that are n-fold (partial) products of `nat`. That is, for any object `x : CompsNR` we have a proof that there exists an `n: nat` satisfying this property that `x = nthProdC ... Par_isCRC nat n`. However, we cannot explicitly select such an `n` based solely on a proof of its existence outside the context of *proving a proposition* because Coq does not allow this type analysis.

To be able to pick a specific `n` for a given `x : CompsNR` as described above, we then require a (type-specific) axiom of choice, specific to only the `Par_isCRC` category and the set `nat` in this context. First, we give the header of the term that, given an object `x : Par_isCRC` and a proof that `x` is of the form `nthProdC ... Par_isCRC nat n` for some `n`, evaluates to a natural number:

```
Definition AC_select_Product (x: Par_isCRC) (pf_prod: exists (n: nat),
x = @nthProdC ... Par_isCRC nat n ) : nat.
```

This term supplies us with a natural number for a given `X: Par_isCRC` which we cannot define directly. Now we must ensure that this natural number satisfies the required property for which it has been selected, i.e., the given object `x : Par_isCRC` is indeed the same object as `nat` raised to the power selected by `AC_select_Product`. This is formalized by the axiom:

```
Axiom AC_Prod_rewrite: forall (x: Par_isCRC)
(pf_prod: exists (n: nat), x = @nthProdC ... nat n ),
x = @nthProdC ... Par_isCRC nat (AC_select_Product x pf_prod).
```

Using this axiom, we can easily prove that selecting a natural number for a given object in the subcategory of n-fold products using this method, then using this number to build an object in this subcategory gives us back the original object:

```
Lemma re_build_obj: forall b, build_compsNR_obj
(AC_select_Product (proj1_sig b) (proj2_sig b) ) = b.
```

The computability predicate `Comp_mapsN` is the formalism needed to finally define the category of objects of the form $\mathbb{N}^n$ and computable maps between them. We define this category as a wide subcategory of the category of n-fold products of `nat` and all partial maps between them (which we called `CompsNR`) using `Comp_mapsN` predicate:

```
Definition CompN: Category.

apply (Wide_SubCategory CompsNR Comp_mapsN).
```

Recall the discussion about instantiating the subcategory of all total maps of a given a restriction category `RC` from Section 5.2. We again use the apply tactic with `Wide_SubCategory`. This tactic reduces the original goal of type `Category` to two subgoals. The first subgoal is a proposition that is true when the identity map in `CompsNR` is computable (i.e. `Comp_mapsN (id a)` it true for all `a : CompsNR`). The identity is expressed in the `prf` language by first taking the n-fold product of projection maps onto each of the n coordinates, i.e., informally, $1_a = \langle \pi_1, ..., \pi_n \rangle$. Then, each of the projections $\pi_i : \mathbb{N}^n$ is computable by `Proj i : prf`. The second subgoal is a proposition which shows closure under composition. Composition in the `prf` language is defined in Zammit's work [50], Section 4.3

We now move on to defining cartesian restriction structure in the `CompN` category we just defined. As in the case of the full subcategory of n-fold products of $\mathbb{N}$, this structure is inherited from `Par_isCRC`. However, all the maps involved (i.e. $\overline{(-)}$, $\langle -, - \rangle$, etc.) must be proved to be contained in this subcategory. That is, we must show each of these is Kleene-equal to some map `f: Hom a b` (for the relevant `a b : CompsNR`) such that `Comp_mapsN a b f` is true. Note that all the proof obligations required for the instantiation of the `RestrictionComb, ParProd` and `ParTerm` type classes with the `CompN` category are automatically satisfied with proofs inherited from the structure in the `Par_isCRC` category.

The proofs of computability of the $\langle -, - \rangle$, π_1 and π_2 maps, as well as the domain map (characteristic function of the subset of $\mathbb{N}$ on which a given partial recursive map

f is defined) are well known results with straightforward informal proofs. In the formal setting, the multiple layers of formalisms involved, the hierarchy of subcategories and coercions, the computations, the destructing of type classes and structures, axiom of choice applications, and the difficulty of parsing the goal syntax are some of the obstacles to the formal study of partial recursion and its abstract categorical model with this approach. The completion of these proofs is also left for future work, and we currently consider them as 'axioms', using '`Admitted`' in our code. Since the explicit definition of the structure is inherited, however, we chose to proceed relying on this existing structure for further formalization.

We have shown, however, that the expected behavior of the restriction combinator in the computable map subcategory formally coincides with the restriction structure inherited from `Par_isRC`. That is, the domain predicate `P: nat → Prop` representing the domain of definition of a partial recursive function,

```
f :  Hom (build_compsNR_obj 1) (build_compsNR_obj 1)
```

is expressible in terms of the `converges_to` predicate in the way we would expect. The predicate `P x` is provable whenever we can demonstrate the existence of a natural number to which an `f_prf : prf` that computes `f` converges on the input `x :: nil`. In the code, the `domain_compute` term formalizes this way of expressing the domain of `f`.

We have also instantiated the type classes formalizing the restriction combinator, partial terminal object and products with cartesian restriction structure inherited from the larger category, `CompCNRC`. Using this structure, we instantiated the cartesian restriction category of computable maps, `CRC_CompN`.

The goal is to prove the resulting category is a Turing category. We will use the equivalent characterization of Turing categories to do so (see Theorem 2.3.6). This characterization requires the definition of a Turing object, Turing morphism (see Definition 2.3.2), and showing that every object in the category is a retract of

the Turing object. Therefore, we need to demonstrate the existence of a Turing object and morphism. First, recall that the enumeration of partial recursive maps and the inverse operation are bijections between $\mathbb{N}$ and the collection of partial recursive maps. This result has been formalized as part of the work done in [50], Section 6.3. For our work, since we are using a formal language defined exactly the same way to represent partial recursive maps, we have axiomatized this result instead of reproving it:

```
Definition enum_prf (f: prf): nat.

Definition nat_to_prf (n: nat): prf.

Axiom nat_prf_nat: forall (n: nat) , (enum_prf (nat_to_prf n)) = n.

Axiom prf_nat_prf: forall (f: prf) , (nat_to_prf (enum_prf f)) = f.
```

The goal now is to Gödel number the partial recursive functions `f : prf`. Given an `f: prf`, the natural number `enum_prf f` is the Gödel number of `f`. Given a natural number `n`, the term `nat_to_prf n` is the `prf` with the Gödel number `n`. The two axioms state that `enum_prf` and `nat_to_prf` are inverses of each other, and thus define a bijection. We give the idea of the definition of this bijection. The two directions of the bijection are fixpoint definitions. In Zammit's original work [50], they are denoted by γ (in our code, `enum_prf`), and $\mathcal{P}$ (in our code, `nat_to_prf`). For brevity and ease of reading, we describe them as follows :

γ: `Zero` $\Rightarrow 0$

`Succ` $\Rightarrow 1$

`Proj i` $\Rightarrow$ i*4 + 2

`Sub f g n m` $\Rightarrow \xi(\gamma(f), \gamma(g), n, m) * 4 + 3$

`Rec f g` $\Rightarrow \pi(\gamma(f), \gamma(g)) * 4 + 4$

`Min f` $\Rightarrow \gamma(f) * 4 + 5$

$\mathcal{P}$: 0 $\Rightarrow$ Zero

$$
\begin{array}{lll}
1 & \Rightarrow \text{Succ} & \\
\text{n+2} & \Rightarrow \text{Proj d}, & \text{if m=0} \\
 & \Rightarrow \text{Sub } \mathcal{P}(\xi_1^{-1}(\text{d}))\ \mathcal{P}(\xi_2^{-1}(\text{d}))\ \xi_1^{-1}(\text{d})\ \xi_2^{-1}(\text{d}) & \text{if m=1} \\
 & \Rightarrow \text{Rec } \mathcal{P}(\pi_1^{-1}(\text{d}))\ \mathcal{P}(\pi_2^{-1}(\text{d})), & \text{if m=2} \\
 & \Rightarrow \text{Minl } \mathcal{P}(d), & \text{if m=3}
\end{array}
$$

where $d = \text{div}(n+2, 4)$

$m = \text{mod}(n+2, 4)$

Here, the terms ξ and π denote computations defined for the $\mathbb{N}^2 \to \mathbb{N}$ embedding (see [50], Section 6.2). We use the bijective property of these mappings in the definition of the applicative (bullet) map in the cartesian restriction category of all computable $\mathbb{N}^n \to \mathbb{N}^m$ maps, `CRC_CompN`. We use the `prf` language to define this map. It is declared as

```
Definition bullet:  Hom (RCat_HP N_obj N_obj) N_obj.
```

Note that as with every Turing morphism, `bullet` must necessarily itself be in the category for which the object $\mathbb{N}$ is Turing. This condition will automatically be satisfied by defining `bullet_CompN` to have the type indicated above (i.e a map between objects in the computable map category, `CRC_CompN`) because maps in this category are sigma types, where the first term is a map in the full subcategory `CompNCRC` of `Par_isCRC` which has type, informally, $\mathbb{N}^n \to \mathbb{N}^m$, and the second is a proof that this map is computable.

Here, we call the term corresponding to `nat:  Par_isCRC` in the `CRC_CompN` subcategory of `Par_isCRC` by `N_obj`. We first define the bullet map, which we call `bmap`, in the underlying category `Par_isCRC`. For each `x :  nat` in the second coordinate of the $\mathbb{N}^2$ product and `prf_f:  prf` corresponding to the natural number in the first coordinate (via the `nat_to_prf` mapping), we select a natural number `y:  nat` such that `converges_to f (x::  nat) y`. This is done via the axiom of

choice selector `AC_select_y`. The domain predicate for this partial map is defined accordingly, in terms of the existence of such a `y`.

Next, we require a proof that this map is indeed computable, i.e. contained in the `CRC_CompN` category. That is, we must show that the definition of the computation which converts a natural number in the first coordinate of `nthProdC ... nat 2` to the partial recursive map corresponding to that number, then applies it to the natural number in the second coordinate, is expressible in the `prf` language. A well-know construction based on Kleene's T-predicate is the universal recursive function, which computes the map described above (see [20], Theorem 4.3, 5.1.2). We have left the explicit definition of such a partial recursive map for future work. The existence of a universal partial recursive map expressed in the `prf` language is formulated as follows :

```
Definition ex_univ_all : exists univ_prf, forall z1 z2 y,
converges_to univ_prf (z1 :: z2 :: nil) y ↔
converges_to (nat_to_prf z1) (z2 :: nil) y.
```

In order to prove this is indeed a Turing morphism in the `CRC_CompN` category, we must define an index `h : Hom N_obj N_obj` for each computable map `f : Hom (RCat_HP N_obj N_obj) N_obj` as required in Definition 2.3.2. For this, we require the S_n^m theorem. In our code, the S_n^m theorem for $m = 1, n = 1$, proved for the `prf` language in [50] (Section 6.3) for all m and n, is stated as an axiom `Smn_11`.

The `s` combinator (as prescribed by the S_n^m theorem) allows us to define the index `h` directly for a computable map `f : Hom (RCat_HP N_obj N_obj) N_obj`. Recall from the definition of the computability predicate `conv_to_cat_prop` that when `f` is computable, there exists an `f_prf : prf` that computes it. Furthermore, we can find the Gödel number (code) `e` of `f` by applying `enum_prf` to `f_prf`. Informally, we write, $\phi_e = f$ (see Example 2.3.5). Now, in the (informal) Turing structure sense, an index `h : Hom N_obj N_obj` of `f` maps a natural number $x \mapsto \phi_s(e, x)$. Here it is

important to note that in Coq, when constructing Turing structure explicity, a proof of the existence of a code `e` of the map `f` does not allow us to directly select the `e : prf` which computes the map. For this reason, we again require another version of the axiom of choice in order to pick the code of a given computable map. In our formalization, we call this `pick_prf`. The term `pick_prf`, given a map $f : \mathbb{N}^n \to \mathbb{N}^m$ as well as a proof that there exists an `e : prf` which computes `f`, and provides us with a specific such code, as well as a proof that this code computes `f`.

We have used `Smn_11` as well as the axiom of choice version `pick_prf` to define an index of the map `f`. In order for it to be a map in the category `CRC_CompN`, we must also show that an index is computable. We have shown that a prf which computes an index of a map `f` with code `e` is a prf which corresponds to the map with the code $\phi_s(s, e)$. Furthermore, as required in the definition of Turing structure, we must show that an index of a map `f` is a total map. This follows from the definition of `h`, since ϕ_s is total. The definition of an index `h` of a map `f`, as well as the proof that is it total computable, is given in our formalization by the term with the header

```
Definition index_map :  forall (f :  Hom (RCat_HP N_obj N_obj) N_obj),
{h :  Hom N_obj N_obj | TotMaps rcCompN _ N_obj N_obj h }.
```

Finally, in order to demonstrate that the `bullet` map defined above, as well as the index maps for each of the computable maps `f : Hom (RCat_HP N_obj N_obj) N_obj` indeed constitute Turing structure in the `CRC_CompN` category, we must demonstrate that the map `f` factors via the `bullet` map, informally $f = \bullet\langle h\pi_1, \pi_2\rangle$. We have done so in the term we call `index_map_commutes_Par`. However, we have only completed this proof for these maps as inhabitants of the underlying cartesian restriction category `Par_isCRC`, leaving the rewriting of sigma types for future work.

Let us also consider using the k, s combinator characterization of combinatory completeness of the `N_obj, bullet_CompN` PCA. While these are rather straightforward to define informally, a formal definition, again, requires the same kind of

machinery that the proof of the S_n^m theorem requires — that is, converting the first natural number in a list into a `prf` (using the `nat_to_prf` mapping) which is a constant function that always outputs this number. The s combinator would also be defined using the `nat_to_prf` term. The level of complexity does not make this approach to proving combinatory completeness any more appealing than a direct proof of the S_n^m theorem, which is why we have not pursued this.

Our formalization of the definition of computable maps formally conforms to the Turing category model, with every map in the category `CRC_CompN` factoring via the `bullet` map and a total (moreover, primitive recursive) map in this cartesian restriction category.

6.2.3 Use of Specialized Versions of the Axiom of Choice

In this chapter as well as the preceding one, we have been faced with the decision of whether to add (weaker versions of) the axiom of choice. In the formalization of categorical concepts in Chapter 4, we did not at any point require adding the AC to construct the necessary formalisms and prove results about them. In this chapter, however, we describe code (found in the **CompN_Cat.v** file) to which we could not avoid adding weaker versions of the axiom of choice in order to define necessary structures and prove key results. The general reason for this is the need to explicitly define all relevant objects and morphisms when building a formal example of a structure described by category theory. For contrast, when proving general category-theoretic results, in the majority of cases it suffices to prove propositions about the nature of the structure being studied.

The following is a summary of the versions of the AC we have added:

(i) `AC_select_y`

— this version, given a `l : list nat` and `f : prf` and a proof that there exists a `y : nat` such that, informally $f(l) = y$ (with l considered as a tuple

of natural numbers in the list `l`), picks the specific `y` which satisfies this property $(f(l) = y)$

— this axiom is required in order to be able to treat the partial recursive maps, presented as relations (where each relation of type `list nat → nat → Prop` holds for a unique `x:nat`) as a partial map any subcategory of `Par_Cat`

— this type of reasoning is essential for all proofs about commuting diagrams in the formalization of Comp(ℕ) we have built

(ii) `AC_select_Product`

— this version, given an object `a : CompNCRC` (in the full subcategory of Par of objects of the form $\mathbb{N}^n$), picks the specific n such that, informally, $a = \mathbb{N}^n$

— this axiom is required for all proofs where we must reason about a map into A^n, by cases, as a map into either $!_A$, A, or a product of maps into A and A^{n-1} (i.e. most inductive proofs about maps in Comp(A)).

(iii) `pick_prf`

— this version, given an object `a b : CRC_CompN` and a map `f : Hom a N_obj` (in the category of computable maps $\mathbb{N}^n \to \mathbb{N}^m$), returns the `f_prf : prf` which is Kleene-equal to `f`, i.e. computes it

— this axiom is needed to be able to use the semantics of the `prf` language, given by the `converges_to` relation, to formally prove the commutativity of the diagrams (including those related to Turing structure) in the category of partial recursive maps between objects of type $\mathbb{N}^n$

In each case, what we have added is, more specifically, an axiom stating a version of the existence property (see [34], Chapter 18, part 2). In (i), the existence property we are axiomatizing is additionally parametrized by proofs of existence of a natural

number y such that $f(l) = y$, as we are dealing with partial recursive maps, and we may only pick the required y when we have a proof that $f(l)\downarrow$. Note that a general version of the existence property is classically false, since it would contradict Gödel's Theorem that Peano Arithmetic is undecidable. In our formalization, however, we are using intuitionistic logic, so it does not introduce an inconsistency.

6.2.4 Partial Results in Defining Ranges in a Formal PRF Category

Considering the approach to the definition of the domain of a partial recursive map that seemed to work best for the purposes of defining the restriction combinator in `CRC_CompN`, studying the (apparently symmetric) definition of range in such a category is the next logical step. We begin by defining a range category of computable $\mathbb{N}^n \to \mathbb{N}^m$ maps,

```
Definition Range_CompN: @RangeCat CRC_CompN ...  rrc_in_CompN.
```

We have defined the range combinator necessary for this instantiation, `rrc_in_CompN`. This range combinator definition, just like cartesian restriction structure in this category, is inherited from `Par_isCRC`. We have omitted the proof that the range of a map in this category is computable (it involves a non-trivial construction using the `prf` language). However, recall that if it exists, in a category where **RR.5** holds (as it does in `Par_isCRC`), the range combinator must be unique, see Section 3.1. Therefore, we know this is the correct choice of range structure.

What we did show, however, is that the predicate `P: (build_compsNR_obj 1) → Prop`, representing the range of definition of a partial map `f: Hom (build_compsNR_obj 1) (build_compsNR_obj 1)`, again (similar to the case of the restriction combinator), is expressible in terms of the existence of a list `x:: nil` such that `f` converges to the given `y: nat` on this list. We call this Lemma `range_compute` in our code. That is to say, the domain and range predicates for the $\mathbb{N} \to \mathbb{N}$ case are symmetrical

in the description of traditional computation using this formalized categorical model, in the following sense: in the case of the restriction combinator, we say that for all `x: nat`, the (inherited) restriction predicate for the map `f` is provable whenever there exists a corresponding `y: nat` to which `f` converges on `x:: nil`. In the case of the range combinator, we say that for all `y: nat`, the (inherited) range predicate is provable whenever there exists a corresponding `x:: nil` to which `f` converges. We expect this result to be formally provable for `f: Hom (build_compsNR_obj n) (build_compsNR_obj m)` for all `n, m : nat`.

In general, the formulation of the axiom of choice `AC_select_y` could have been done, instead, in terms of selecting a list `ln` for the given `f, y` and the proof of existence of a list satisfying the required `converges_to f ln y` proposition. In the context of our formulation of the `prf` language, we would again be unable to deterministically search for a list that satisfies the required proposition as there is no way to formally do this without a guarantee that the computation will terminate, and therefore such a computation cannot be implemented in Coq. The reason for this symmetry is that we are, in effect, making a subcategory of sets and relations conform to a partial function categorical structure.

Chapter 7

Conclusion

7.1 Discussion of Contributions

7.1.1 Formalizing Abstract Computational Structure

The discussion in this section is about the results presented in Chapter 5, which describes the work we have done to complete contributions (i)-(iii) listed in the introduction, as well as a reflection on items (vii) and (ix). We will now examine some of the hurdles of this part of the formalization process, as well as some observations about it. Formalizing some of the categorical structure required for reasoning about Turing categories was reasonably straightforward. In particular, formalizing cartesian restriction structure closely followed the format that was set in the existing category theory library. However, proofs about restriction products and restriction terminal objects have been significantly complicated as compared to (true) products and terminal objects proofs by the addition of restriction structure.

In general, it is rather tedious to deal with categorical structures as collections of terms (including proof obligation terms), rather than the objects or maps they are normally represented by informally (e.g. just the object $A \times A$ rather than the corresponding type class). The situation becomes complicated by having to consider

all terms within the set of proofs and type definitions that are referenced within the set of hypotheses (i.e. nested sigma types), as well as the terms in the goal itself. When rewriting a goal containing sigma-type terms, one might ignore, for example, the types of the proofs within a term of a sigma-type and attempt a rewrite that will generate a goal containing an ill-typed term. However, Coq has a vigilant type-checking system and does not allow this to happen, generating an error instead.

In addition, a formal study of Turing categories and related categorical structure required a lot of reasoning about structure within a subcategory hierarchy defined in terms of predicates on objects or maps (e.g. the total subcategory in Section 5.2), as well as boundary cases (e.g. formal n-fold product definition in Section 5.5.1), which are both inconvenient to work with. Subcategories are defined in terms of predicates on objects or morphisms of the larger category, and are therefore necessarily sigma types. Boundary (or degenerate) cases, such as the 0-fold product of the natural number object (with itself), are sometimes not the same as the general case (e.g., in this case, a terminal object rather than a product). We may conclude, then, that dependent types are an important formalism in CIC, responsible for much of its expressive power. Our success in building the desired categorical formalisms relied heavily on this feature. Sigma-types, however, are also in many cases what makes it rather difficult to work with formal categorical structure.

7.1.2 Formalizing Categorical Examples

Chapter 6 of this thesis describes our approach to formalization of categorical examples of structures that must exist (or, in the case of range structure, can exist, see Section 6.2.4) in a Turing Category. The examples we chose were the motivating examples of each type of structure, and their formalization corresponds to items (iv) and (v) in the contributions list. We now present a discussion of items (vi) and (viii), which draw conclusions about these formal examples. One of the major challenges we

faced in the formalization of examples, which does not arise in the informal setting, is building concrete examples of constructs involving partiality. The reason for this is that when a map's domain of definition is "smaller" than the origin object, this would need to be reflected in the formal language description of the function as an infinite computation. Alternatively, one may attempt to model partiality using pointed sets (as discussed in Section 5.1). However, the concept of mapping a function which is undefined on a given input to $\downarrow$ does not correctly model the concept of 'does not halt'. The reason for this is that in intuitionistic logic, it may not be provable that a given map does not halt on some input. For this reason, an (informal) partial map must be represented by a different (related) map. The domain of this new map must be built into the definition in such a way that it restricts the application of this the map to only the elements of the set in its domain. This requires a lot of re-structuring of the informal definitions in order to fit this formal approach.

This feature of formal reasoning about partial maps extends, not surprisingly, to formalizing all subcategories of the category of sets and partial functions. Therefore, when reasoning about partial recursive maps, we are forced to first consider such maps as relations, as it is impossible to define an arbitrary prf's domain predicate explicitly (not in terms of the map itself, see Section 5.2 for a discussion of this topic). Encoding prf's as relations betrays the applicative nature of what we usually think of as computation. Traditionally, computation is considered as an input-output model. We followed our convictions that staying true to this description in our formalization is the best way to gain insight into how a traditional and a generalized categorical computation model could be formally integrated. We have done so by skewing the inherent symmetry of a relation-based encoding with the use of the axiom of choice, which selects an output whenever a prf is defined on a given input, in order to represent partial functions (and so, also the input-output model). This way, we have also aligned the partial recursive map formalization approach with the formalization of the partial structure in the underlying category Par. We decided to use the predicate

which relates a prf with its input and output pairing as the domain of the given prf to build the corresponding partial map. This approach may seem redundant, but it is a reliable way to make sure we do not produce any 'undefined computations' (while still describing partial recursion), which, in reality, do not exist within the category.

Defining the traditional computation example, we have witnessed how exactly the prf formalism can be expressed in the language of category theory. We have also verified that the resulting category constitutes a formal case study of the idea that to perform computation, an object must be representative of its own (partial) function space (as modeled by Turing structure). And finally, we have formally demonstrated the close (symmetric) relationship between the domain and the range of partial recursive function. Having conducted this formal language investigation into describing what is ordinarily known as 'formal' computation, it becomes apparent that there are a lot of nuances overlooked when reasoning about computation informally. One of the biggest obstacles in translating informal concepts, especially those related to computation, into a formal language, is that there is no possible set of instructions on how to do so (would these have to be formal or informal?), or theorem to prove the correctness of this translation. In this work, we have developed a formalization of (traditional) computation as it is modeled by a Turing category. We believe that this formalization not only correctly portrays the concepts of the category and recursion theories (as well as the formal way to describe one via the other), but also makes precise all the details of the corresponding informal definitions and results.

7.1.3 Concluding Remarks

In this section we present a general discussions on how our research reflects item (x) of the list of contributions listed in Section 1.6. The motivation for undertaking this project can be summarized as follows: seeing as computation on a physical computer is a formal process, it is worthwhile to work towards a machine encoded and checked

proof that a (category-theoretic) construct does indeed constitute a model for this phenomenon. This project accomplishes this in two general parts: formalization of general categorical structure (described in Chapter 4), and formalization of specific examples of this structure (described in Chapter 5). Each of these presented its own set of insights and challenges (discussed above). We may summarize the merit our project in the following way:

(i) Developing a tool which uses a formal language to reason about the nature of an abstract model of computation

(ii) Using this tool to build a hierarchy of cartesian restriction subcategories of Par, with the smallest being category of partial recursive maps

(iii) Modeling partial computation (by means of a relation) using a formalism which allows only total computation, then constructing partial recursive maps which correspond to those expressed by this relation, and

(iv) Using different proof techniques to manipulate the resulting structures

— i.e. while the underlying high-level proof idea is similar, the structure of formal terms is different from their informal counterparts and therefore we must manipulate it in a different way in a formal proof

(v) Highlighting the difference between formal classical and constructive reasoning about partiality

(vi) Revealing omissions in the informal definitions, proofs and propositions, examples of which include:

— Given a cartesian restriction category C, with $A \in \mathsf{C}$, stating the correct conditions for $\mathrm{Split}(\mathrm{Comp}(A))$ to be a cartesian restriction category

— Reformulating a result about ranges in Turing categories in terms of open maps instead of the range combinator

(vii) Laying the groundwork for formally studying abstract computation, integrated into a larger formalization project with many opportunities and directions for expansion

A notable advantage of expressing concepts related to computability formally is that these formalisms are robust in the sense that they conform to a strict hierarchy and satisfy precisely defined relationships within all the structure involved, as opposed to the often ad-hoc approach of traditional computation. This allows us to structure existing results on computability in a way that could facilitate and potentially even automate future research on the topic. The existing formalization of the structure and key properties of computable maps has allowed us to formalize a selection of different concepts built on top of basic category and recursion theory and conduct a study that truly focuses on categorically modeling computation.

An important goal of this formalization project is to conduct an investigation into whether the chosen formal language is powerful enough to describe structure within a generalized theory of computation. We see that the CIC, while not ideal for this task in terms of convenience, has stood up to the task of encoding the language of Turing categories. We believe that in any further formalization of Turing categories, examples thereof, and related structure, it would be worthwhile to proceed using the CIC language. It is an extensively developed, documented and used language describing a powerful logic — hence, it is a good choice for the purpose.

This formalization project does not constitute a stand-alone library covering all aspects of a given theory or application; however, because of this, it enjoys all the advantages of a formalization project integrated with an existing library which encompasses a large set of the definitions and theorems of the theory of categories. This project is a contribution to a larger integrated effort of formalizing category theory.

Not starting from scratch (using the existing category theory library as well as an existing formalization of partial recursive maps), this thesis establishes the foundation for formalizing the study of abstract computability using the Turing category model.

Formalizing Turing categories and related concepts is a way to verify the validity, versatility, usefulness and limitations of the Turing category model as it relates to existing computational systems, and lets us know that we are taking the correct approach of studying abstract computability. This formalization project has yielded interesting insights into the application of Coq in the domain of computability, including the practicality of formalizing computability using this proof assistant (as discussed above). It highlights the differences in approaches to reasoning about recursion in an informal way and those which are formally correct and is able accommodate a variety of directions for further expansion of the project.

7.2 Future Work

While this formalization project builds a solid framework of the structures to study abstract computability formally, as well as the relationships between them, there are a number of promising directions for expanding this formalization both in terms of breadth and depth. A natural expansion of this formal study of Turing categories is to formalize the motivating categorical examples (i.e. sets and partial maps and its subcategories) by handling Kleene equality as a setoid rewrite. Another natural way to expand on the work done in this project is to formally define all the isomorphisms between n-fold products of the set of natural numbers with itself (and possibly the terminal object) associated in all possible ways, as well as the coercions necessary for the applications of these isomorphisms to be automated. Yet another desirable extension of this project is to define Kleene's T-predicate, and use it to formalize the universal partial recursive function.

Perhaps one of the most promising formalization directions is to build formal

definitions of the categorical structures necessary to establish abstract versions of well known recursion theory results in a Turing category (such as Rice's theorem) that reduce to their traditional versions in the Comp(N) case [12]. This is an attractive prospect because it eliminates the need to reason about set maps extensionally, and allows using restriction structure to study recursion, instead of the cumbersome formalization of set map partiality using a strongly normalizing formal language. Furthermore, recent work establishes criteria for determining when various complexity classes of total maps (i.e. PTIME, PRIM, etc.) can be made into a Turing category [11] and studied abstractly, which provides motivation to transition to studying formal computation in a formal abstract setting, rather than as (informally-presented) and set-based.

For other directions of future work, one option is to formalize further results specifically about Turing structure, or perhaps monoidal Turing categories instead (eventually, with differential structure). Another option is to conduct a formal study more focused on the PCA's (which, recall, are computation-modeling structures at the core of every Turing category) and relationships between them, or consider a different model of partial computation to formalize, such as cartesian closed restriction categories.

Another avenue to explore for future formalization is to formally add other types of structure to a Turing category. We have formalized range structure in a Turing category, but one may also consider coproducts, the equality map. It could also be an interesting study to formalize the logic rules of the computational systems and formally verify their correctness, at the same time exploring the process of converting a categorical abstraction into a logical system.

References

[1] Steve Awodey. *Category Theory*. Oxford Logic Guides. Ebsco Publishing, 2006.

[2] Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[3] Christoph Benzmüller and Dana S. Scott. Axiomatizing category theory in free logic. *Computing Research Repository*, abs/1609.01493, 2016. `http://dblp.uni-trier.de/rec/bib/journals/corr/BenzmullerS16`.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[5] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.

[6] Pierre Castéran and Matthieu Sozeau. A gentle introduction to type classes and relations in Coq, May 2012. `https://hal.archives-ouvertes.fr/hal-00702455`.

[7] James Chapman, Tarmo Uustalu, and Niccol Veltri. Formalizing restriction categories. 2017.

[8] James Chapman, Tarmo Uustalu, and Niccolo Veltri. The delay monad and restriction categories. *NWPT 2013*, page 22.

[9] Arthur Charguéraud and François Pottier. Proceedings of the interactive theorem proving: 6th international conference. pages 137–153. Springer International Publishing, 2015.

[10] A. Church. *The Calculi of Lambda Conversion. (AM-6).* Annals of Mathematics Studies. Princeton University Press, 2016.

[11] J.R.B. Cockett, P.J.W. Hofstra, and P. Hrubeš. Total maps of turing categories. *Electronic Notes in Theoretical Computer Science*, 308:129 – 146, 2014. Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXX).

[12] Robin Cockett. Turing categories and computability. Slides from 15th Estonian Winter School in Computer Science (EWSCS), 2010.

[13] Robin Cockett, Xiuzhan Guo, and Pieter Hofstra. Range categories II: Towards regularity. *Theory and Applications of Categories*, 26(18):453–500, 2012.

[14] Robin Cockett and Pieter Hofstra. An introduction to partial lambda algebras. February 2007.

[15] Robin Cockett and Pieter Hofstra. Introduction to Turing categories. *Annals of Pure and Applied Logic*, 156(2-3):183–209, 2008.

[16] Robin Cockett and Stephen Lack. Restriction categories I. *Theoretical Computer Science*, 270:223–259, 2002.

[17] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121(12):89 – 112, 1993.

[18] The Coq Development Team. *The Coq Reference Manual, version 8.5*, January 2016. `https://coq.inria.fr/distrib/current/files/Reference-Manual.pdf`.

[19] Thierry Coquand and Gérard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.

[20] Nigel Cutland. *Computability: An introduction to recursive function theory*. Cambridge University Press, 1980.

[21] Peter J. Freyd and Andrej Scedrov. *Categories, Allegories*. North-Holland Mathematical Library. Elsevier, Amsterdam, 1990.

[22] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[23] S. Goncharov. *Countable Boolean Algebras and Decidability*. Siberian School of Algebra and Logic. Springer US, 1997.

[24] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382 – 1393, 2008.

[25] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. *A Machine-Checked Proof of the Odd Order Theorem*, pages 163–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[26] Paul R. Halmos. *Naive Set Theory*. Undergraduate Texts in Mathematics. Springer New York, 1998.

[27] Pieter Hofstra and Jaap van Oosten. Ordered partial combinatory algebras. *Math. Proc. Cambridge Philos. Soc.*, 134(3):445 – 463, 2003.

[28] Gérard Huet. Tapsoft 1987: Proceedings of the international joint conference on theory and practice of software development. pages 276–286, 1987.

[29] Gérard Huet and Amokrane Sabi. Constructive category theory. In *Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*. MIT Press, 1998.

[30] André Joyal, Mogens Nielson, and Glynn Winskel. Bisimulation and open maps. *Proceedings of Eighth Annual IEEE Symposium on Logic in Computer Science*, LICS '93:418 – 427, 1993.

[31] Alex Katovsky. Category theory in isabelle/hol, 2010.

[32] John L. Kelley. *General Topology*. Graduate Texts In Mathematics. Springer, Birkhauser, 1975.

[33] S.C. Kleene and M. Beeson. *Introduction to Metamathematics*. Ishi Press International, 2009.

[34] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 7 edition, 1986.

[35] G. Longo and E. Moggi. *Gödel numberings, principal morphisms, combinatory algebras*, pages 397–406. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984.

[36] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Monographs in Computer Science. Springer New York, 2012.

[37] Russell O'Connor. *Incompleteness and Completeness: Formalizing Logic and Analysis in Type Theory*. PhD thesis, Institute for Computing and Information Science, Faculty of Science, Radboud University Nijmegen, 2009.

[38] Greg O'Keefe. Towards a readable formalisation of category theory. *Electronic Notes in Theoretical Computer Science*, 91:212–228, 2004.

[39] Erik Palmgren and Olov Wilander. Constructing categories and setoids of setoids in type theory. *Logical Methods in Computer Science*, 10(3:25):1–14, 2014.

[40] Robert A. Di Paola and Alex Heller. Dominical categories: Recursion theory without elements. *J. Symbolic Logic*, 52(3):594–635, 09 1987.

[41] Lawrence C. Paulson. A mechanised proof of Gödel's incompleteness theorems using nominal Isabelle. *J. Autom. Reason.*, 55(1):1–37, June 2015.

[42] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[43] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2015. `http://www.cis.upenn.edu/~bcpierce/sf`.

[44] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations program, 2013.

[45] Amin Timany and Bart Jacobs. Category theory in Coq 8.5. In *The 7th Coq Workshop*, Sophia Antipolis, France, June 2015.

[46] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[47] Polina Vinogradova. Investigating structure in turing categories. Master's thesis, University of Ottawa, March 2012.

[48] Vincent Zammit. A mechanisation of computability theory in HOL. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '96, pages 431–446, London, UK, UK, 1996. Springer-Verlag.

[49] Vincent Zammit. A comparative study of Coq and HOL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '97, pages 323–337. Springer-Verlag, 1997.

[50] Vincent Zammit. A proof of the S-m-n theorem in Coq. Technical report, The Computing Laboratory, The University of Kent, Canterbury, Kent, UK, March 1997. `http://kar.kent.ac.uk/21524/`.