Partial Functions and Recursion in Univalent Type Theory

by

Cory Knapp

A thesis submitted to
The University of Birmingham
for the degree of
Doctor of Philosophy

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
June 2018

Abstract

We investigate partial functions and computability theory from within a constructive, univalent type theory. The focus is on placing computability into a larger mathematical context, rather than on a complete development of computability theory. We begin with a treatment of partial functions, using the notion of dominance, which is used in synthetic domain theory to discuss classes of partial maps. We relate this and other ideas from synthetic domain theory to other approaches to partiality in type theory. We show that the notion of dominance is difficult to apply in our setting: the set of Σ^1_0 propositions investigated by Rosolini form a dominance precisely if a weak, but nevertheless unprovable, choice principle holds. To get around this problem, we suggest an alternative notion of partial function we call disciplined maps. In the presence of countable choice, this notion coincides with Rosolini's.

Using a general notion of partial function, we take the first steps in constructive computability theory. We do this both with computability as structure, where we have direct access to programs; and with computability as property, where we must work in a program-invariant way. We demonstrate the difference between these two approaches by showing how these approaches relate to facts about computability theory arising from topos-theoretic and type-theoretic concerns. Finally, we tie the two threads together: assuming countable choice and that all total functions $\mathbb{N} \to \mathbb{N}$ are computable (both of which hold in the effective topos), the Rosolini partial functions, the disciplined maps, and the computable partial functions all coincide. We observe, however, that the class of all partial functions includes non-computable partial functions.

Acknowledgements

As always, there are far too many people to thank, and I'm bound to miss some, so I'll just leave it at the obvious, and academic: Thanks, especially, to my supervisor Martín Escardó, for patient explanations, sage advice, and long working meetings. I cannot express how helpful of a supervisor you have been. Thanks to the staff in the Birmingham theory group—especially to Paul Levy, for teaching me category theory faster than I knew it could be taught, and for consistently playing devil's advocate; to Achim Jung, for explaining logical relations to me; and to Steve Vickers, for explaining geometric reasoning. Thanks also to the PhD Students in the group—especially to Auke for answering all my mathematical questions, and to everyone involved in CARGO for continuous motivation.

Thank you to the examiners Steve Vickers and Bernhard Reus for taking the time to read this.

Mike Shulman and Peter Lumsdaine may not know it, but they helped set me on the path that led here: one by being kind to a clueless undergrad, and the other by giving direction to a masters student in over his depth. Both have also given advice and encouragement since.

Finally, of course, I couldn't have done this without endless support from friends and family, but such thanks are better said in person.

Contents

Introduction

In *Mathematics as a Numerical Language* [10], Bishop outlines what he viewed as the most important questions of constructive mathematics of his time. In it, he states "that mathematics concerns itself with the precise description of finitely performable abstract operations." He also states that constructive mathematicians can use classical mathematics "as a guide" and that much of classical mathematics "will raise fundamental questions which classically are trivial or perhaps do not even make sense." The essay continues by sketching three problems that are classically easy, but when viewed constructively demonstrate gaps in our conceptual understanding. He states "Each of the three problems just discussed requires the development of new concepts appropriate to the constructive point of view. None of them is likely to be given an acceptable solution by the application of a general technique of constructivization." In fact, for two of these problems (Birkhoff's ergodic theorem and the construction of singular cohomology), there is no constructive solution, so a new approach must be taken to develop the field. For the third (Hilbert's basis theorem), the proof itself is constructive, but exhibiting objects to which the theorem can be applied necessarily requires non-constructive arguments.

This thesis is focused on computability theory, but done from the point of view of constructivization that we see in the previous paragraph. That is,

- our mathematical work should describe "finitely performable abstract operations";
- we will be guided by the classical development of computability theory, but
- we will pay special attention to fundamental questions which are classically trivial and on concepts that are appropriate for the constructive point of view.

The goal, then, is not to develop a full constructive account of computability theory but to understand how computability theory fits into a constructive mathematical world. In particular, the behavior of partial functions is more subtle in a constructive world, so we spend a considerable amount of time examining the theory of partial functions.

In order to proceed, we must fix a constructive framework for doing mathematics. Formally, we use a modest extension of intensional Martin-Löf type theory (MLTT). This connects directly to our motivating paragraph: while introducing his type theory in [57], Martin-Löf cites Bishop [10, 9] as a motivation for MLTT. We will not work formally: since we are communicating mathematics, and not describing a computer formalization, we will describe and use this implicit framework in an informal way. Nevertheless, it should be possible to formalize the material in this thesis directly in a proof assistant such as Coq [19] or Agda [61].

Both the approach we take to mathematics and the language we use to communicate it are not widely familiar. The traditional constructive approach is a *Bishop-style* mathematics: one gives sets via a collection of elements and an equivalence relation saying how to identify these elements. Mathematics in MLTT usually mimics this approach using setoids (Section 1.2 and Chapter 6), but more recent work tends to be done from a *univalent* perspective (described in 1.5), which we take here. Briefly, univalent mathematics is an approach to mathematics which instead works with the identity types for equality, and which pays special attention to the structure arising from this proof-relevant equality. This attention motivates several new definitions, in particular a new interpretation of logic in type theory. Moreover, this approach allows us to formally distinguish between notions seen as structure and notions seen as property. Practical work in this approach requires the addition of some extensionality principles to MLTT (Chapter 2).

Part I of this thesis develops the univalent approach to constructive mathematics. The aim is to present a concise, self-contained and readable introduction to univalent mathematics, going somewhat beyond what is required in Part II. Most of the material in Part I can be found already in the HoTT Book [86], but we make a number of different choices in terminology and presentation. In particular, we simplify many proofs, insist less on homotopical intuition, and reorder the priority and primacy of various concepts to better fit with the aims of Part II and the changes in the field over the last several years. We also spend quite some time discussing the relationship between extensionality axioms. There is a technical contribution here: both the proof that univalence implies function extensionality and the surrounding discussion in Section 2.4 are new, although some of the ideas were already implicit in Voevodsky's proof as presented in [33].

Part II begins the main technical contributions of the thesis by examining partial functions. There are several approaches to partial functions (Chapter 4 and Sections 5.1 and 5.2), but the most technically useful is one based on a notion of *partial element* (Definition 5.3), which is a type-theoretic version of the notion of subsingleton subset, as well as a constructive version of the lifting from classical domain theory; indeed, all lifted sets are DCPOs (Section 5.4).

As mentioned above, our broad aim is to better understand how the computable partial functions fit into a constructive world. Although intuition tells us that anything which can be done constructively can be done computably, we can define a partial function $h : \mathbb{N} \rightharpoonup \mathbb{N}$ which enumerates the complement of the halting set, and so cannot be a computable partial function. It is worth asking then which partial functions *can be* computable. Chapter 5 is devoted to shedding light on this question by looking for a restricted notion of partial function for which it is consistent that all such partial functions are computable. Importantly, we expect such partial functions to be composable. The obvious way to find a more restricted notion of partial function is via the notion of *dominance*—set of propositions satisfying certain closure properties (Section 5.5). Dominances were introduced and studied by Rosolini [75] as part of a project of *synthetic domain theory*, and modulo size issues (Section 5.6), the theory of dominances transports directly from a topos-theoretic to a type-theoretic setting. Importantly, we can give a restricted notion of partial function for any set of propositions S, and the partial functions arising this way are closed under composition precisely if S is a dominance. This fact suggests that we can use the dominance of *Rosolini propositions*, typically called Σ, as our restricted notion of partial function. Indeed, this is the approach taken in work on both synthetic domain theory and synthetic computability theory.

Unfortunately, a weak version of countable choice, which is not provable in our system [26], is required to show that the Rosolini propositions form a dominance (Theorem 5.30). The Rosolini dominance arises by truncating a family of types—by replacing a structure with the property that there is such a structure; for any set of propositions S arising this way, there is a weakening of the axiom of choice equivalent to the claim that S is a dominance (Theorem 5.32). From the perspective of partial functions, the issue is that intensional information is used to compose pairs of S-partial functions, but this intensional information is hidden when we truncate S. That is, S-partial functions come with no guarantee that they respect the required intensional information. To resolve this issue we look at a notion of *disciplined map* (Section 5.10), which can be seen as the class of maps which respect the intensional information used to compose S-partial functions. Indeed, disciplined maps can be shown to compose even without choice (Theorem 5.44), while countable choice implies that the disciplined maps out of $\mathbb{N}$ are exactly the S-partial functions from $\mathbb{N}$ (Theorem 5.46). That is, in the settings examined by synthetic domain theorists and synthetic computability theorists, the Rosolini disciplined maps are the Rosolini partial functions, making them a natural candidate notion of computable function.

In Part III, we finally turn to computation. Facts about computation from the topos-theoretic perspective seem to conflict with facts about computation from the type-theoretic perspective. From the topos-theoretic perspective,

1. it is *consistent* that all total functions $\mathbb{N} \to \mathbb{N}$ are computable: this is true of constructive higher-order logic, with the effective topos as a model.
2. It cannot be proven that there is an embedding from the class of computable functions to the natural numbers: this embedding would tell us that excluded middle holds for equality between computable functions, which fails in the effective topos.

On the other hand, from the type-theoretic perspective

1. it is *false* that all total functions $\mathbb{N} \to \mathbb{N}$ are computable: an argument by Troelstra shows that even in a constructive higher-order setting, this would allow us to solve the halting problem.
2. There is an embedding from the class of computable functions to the natural numbers: this is a first theorem in computability theory.

Univalent mathematics gives a general framework incorporating both of the above perspectives. In particular, the topos-theoretic facts correspond to facts about computability as *property*, while the type-theoretic facts correspond to facts about computability as *structure*. We present computability via a notion of *recursive machine* (Section 7.2) which abstracts away details of initialization, state and memory from the definition of Turing machine. We say that a *computation structure* for a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is a recursive machine computing f, while *being computable* is the associated property arising by truncating the type of computation structures. Much of a first course in computability theory goes through for both computation as structure (Chapter 7) and computation as property (Chapter 8). The computability theory we present is not deep, and we reiterate that a complete development is not the goal. Rather, these results should show that our approach to computability theory is sound.

The final technical Chapter 9 contains a discussion of how computability as structure and computability as property differ, and a look at how computability theory fits with the view of partial functions developed in Chapter 5, bringing the two threads back together. The key result is that countable choice and "Church's thesis" (the statement that all *total* functions $\mathbb{N} \to \mathbb{N}$ are computable) together imply that the computable partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$ are exactly the Rosolini partial functions. There is an unfortunate lacuna here: one of our goals is a notion of partial function for which it is consistent that all such partial functions are computable. It is indeed consistent with MLTT that our candidate notion of disciplined map is such a notion: in the effective topos, the disciplined maps, the Rosolini partial functions and the computable partial functions coincide. However, we use a univalent version of proposition extensionality throughout, and this statement relies on treating propositions as types satisfying a certain property, rather than as elements of a sub-object classifier, as is done in topos logic. We would need to either give a translation from our type theory to topos logic, or construct a model validating countable choice and Church's thesis to resolve this discrepancy. For reasons of scope, we do neither here. Filling this gap is one of the obvious directions for future work. Another direction for future work is to examine higher-type computability from the perspective developed here. We discuss both of these directions in more depth when discussing future work.

A few threads woven through this thesis determine its shape, despite being difficult to express in a way that is both general and technical. I will try to explain them here, with reference to Bishop's essay *Schizophrenia in Contemporary Mathematics* [11].

Bishop's essay concerns itself primarily with what Bishop calls the *debasement of meaning*. Two particular trends leading to this debasement (according to Bishop), were the tendency towards formalism—in which meaning is simply ignored, in favor of formal manipulations—and an "esotericism"—in which meaning becomes imprecise. Strangely, despite the particularly formal subject matter, it seems computability suffers more from the latter than the former. For example, many books on computability theory will fix some formal notion of computation (or discuss and compare several), and then after the first chapter, prove all results with an appeal to the Church-Turing thesis and a rough description of an algorithm, leaving the reader left to wonder why they went through the trouble of understanding (e.g.) Turing machines in the first place. The appeal to intuition implicit in the reference to the Church-Turing thesis would be more honest if done as in other areas of mathematics: rather than suggest we are appealing to an empirical principle, simply describe informally how to construct the mathematical object of interest.

For his part, Bishop responds to the identification of finitely-performable and recursive by saying that the "naive concept" of an algorithm is more basic than that of recursive function, and "the recursive concept derives whatever importance it has from some presumption that every algorithm will turn out to be recursive." Bishop, then, seems to take the view that computability theory is a misguided attempt to make more precise the concept of algorithm. I take a different view: computability theory is the mathematization of the real-world concept of program, and programmable functions. This mathematization has proved fruitful in aiding our understanding of programming, and one may suspect, will continue to do so. In other words, computability theory is a part of mathematics which deserves treatment.

A more precise (and less polemical) example of the debasement of meaning in computability theory is given by the notion of *Kleene equality*. One often says that $f(x) \simeq g(x)$ when either both $f(x)$ and $g(x)$ are undefined, or both are defined and equal. This does not work in a constructive setting, since we cannot say that a function is either defined or undefined on some input. Moreover, the treatment of partial functions is such that the terms $f(x)$ and $g(x)$ are only meaningful when defined. That is, this notation makes reference to a non-existent thing, and we explain this notation by reference again to this non-existent thing. The treatment given here differs: we have no need of Kleene equality, because partial functions $f : X \rightharpoonup Y$ are taken to be ordinary function into a special type of partial values of Y. Then, $f(x) \simeq g(x)$ precisely if $f(x)$ and $g(x)$ are equal as partial values.

In this case, an ad hoc notion of equivalence can be reduced to an honest equality by revising our approach. This idea—replacing ad hoc notions of equivalence with the general notion of equality—represents a crucial benefit of the univalent approach. Working directly with setoids, as Bishop does and as is traditional in type theory, brings with it byzantine bureaucracy, arising from the ad-hoc treatment of equality. Martin Hoffman showed how to interpret MLTT with equality in setoids [35], so MLTT can be used to simplify this bureaucracy. Identity types provide a uniform and general way to present the machinery of setoids. However, some *extensionality* principles are required to treat identity types properly, and there has long been a question about how best to approach extensionality in type theory. The univalence axiom provides one possible answer to this question.

The last point concerns the meaning of existence. One of Bishops concluding remarks is "There seems to be no reason in principle that we should not be able to develop a viable terminology that incorporates more than one meaning for some or all of the quantifiers and connectives." A particular point of dispute between classical and constructive mathematicians is the meaning of the existential quantifier. The univalent perspective gives two versions, one which we denote $\exists$ is valued in *propositions*, the other is denoted Σ and is valued in *structures*. While our logic of propositions is not classical, the meaning of $\exists$ seems more in-line with the logicist interpretation used in classical mathematics, while the meaning of Σ aligns with the intuitionistic interpretation used in constructive mathematics. These connectives are made possible by the distinction between properties (families of propositions) and structures (general type families) in a univalent setting. This distinction is certainly not unique to univalent mathematics, but it seems to be the first to give a formal language for exploring the distinction. The work reported here began as an attempt to study the distinction between structure and property, not at a general abstract level, but in the wild, with a particular example. It grew into its present form by returning repeatedly to the central question: how does computability viewed as structure differ from computability viewed as property?

Summary of contributions

The main contributions are Chapters 5, 7, 8, and 9. Some of the material in these chapters has appeared in [31]. Sections 5.8-5.10 contain the truly novel material of Chapter 5; the rest of of the chapter is largely an exercise in translation from one framework to another. The technical results in Chapters 7 and 8 are standard, well-known results, but they have not been proved in a univalent setting, and the key contribution is the clear and rigorous distinction between computability as structure and computability as property. This distinction is what leads to the denouement in Chapter 9. Except for Theorem 9.9 due to Troelstra, this chapter is entirely new. The discussion of partiality via setoids in Chapter 6 is new, but is only in sketch, and serves mostly as comparison.

Part I is standard material. The main technical contribution is the new proof that univalence implies function extensionality in Section 2.4. There are some minor contributions: some minor simplifications in other proofs (the proof of Theorem 1.13 comes to mind), and Chapter 3, which is mostly folklore, but which contains some material that does not seem to be published anywhere. Nevertheless I feel there is a larger contribution here: the organization of concepts in univalent mathematics has changed in the 5 years since the HoTT book was first published. Part I provides an introduction to the field which reflects (some of) these changes.

Part I

Univalent Mathematics

Part I deals with univalent foundations. We begin with an overview chapter which explains the underlying system (Section 1.1), relates this system to approaches to constructive math (Sections 1.2-1.4), and then introduces the univalent perspective (Sections 1.5-1.9). It is important to emphasize that the way we make intuitive ideas rigorous in a univalent setting is different from the classical approach. The first chapter is is chiefly designed to give language that better fits the univalent approach, and to give intuition on how it differs from other approaches to constructive mathematics. In particular, since univalent mathematics is built on a type-theoretic framework, logic is not an underlying part of the language, but something which we must interpret. Instead of the Curry-Howard interpretation taken in traditional type-theoretic approaches, we take a "propositions as subsingletons" view of logic, isolating the propositions as a distinguished class of types. Importantly, this distinction is made internally (not using a judgement), using a defined operator $\mathsf{isProp} : \mathcal{U} \to \mathcal{U}$; except for constructions made using isProp, propositions are treated as any other type.

With this language in place, we develop basic tools for the univalent perspective in Chapter 2, focusing particularly on extensionality principles (Sections 2.1 and 2.4) and on propositions (Sections 2.2). We will see quickly that pure MLTT is insufficient for developing a logic of propositions using isProp. We must extend the type theory with a *truncation* operator, taking a type to its best representation as a proposition. We will also need the propositions to be closed under Σ and Π; Closure under Π happens to be equivalent to extensionality for functions (pointwise equal functions are equal), while closure under Σ is true already in MLTT. Similarly, it is desirable to have an extensionality principle for propositions (logically equivalent propositions are equal). We will assume both of these principles in Parts II and III, but here we discuss the *univalence axiom*, which serves as an extensionality principle for types, implying both proposition and function extensionality. Here, and throughout, we use the term *extensionality principle* to mean a principle which allows us to prove equality between objects from a seemingly weaker relationship. We do not consider the question of whether such principles make the theory truly extensional (for this, see [82] or [50]). Sections 2.5, 2.6 and 2.7 cover ideas that are new to univalent mathematics, via applications to constructions that have been problematic in type theory.

Finally, we examine a few traditional notions from the univalent perspective in Chapter 3. Of particular importance are Section 3.5 on an internal notion of monad on the universe of types, and Section 3.6 on the axiom of choice. These are not of note because the univalent approach is interesting (although with choice, it is), but because we will be particularly concerned with monads and choice principles in Part II.

CHAPTER 1

Type theory

To begin, we explain the base formal system, which we will later extend in Chapter 2. We then move on to a discussion of the traditional way to interpret logic in this system, and a discussion of informal constructive mathematics. To end the chapter, we give a small taste of univalent foundations, discussing the perspective taken in a univalent development and giving the basic definitions needed to explore the univalent approach.

1.1 Martin-Löf type theory

We work in an extension of MLTT similar to that in the HoTT Book [86], and use some language from the HoTT Book. Our theory is a theory of objects called *types* which have members called *elements* or *inhabitants*. We write $a : A$ to mean a is an element of the type A. Judgmental equality–which expresses a syntactic notion of equality—will be denoted by $\equiv$. Elements can only be given as elements of a specified type. That is, we have no global universe of discourse or global membership predicate. We have several basic types and type operations, each of which come with *formation rules* saying how to create the type, *introduction rules* saying what the type's *canonical* elements are, *elimination rules* saying how to use elements of the type, with a specification (the *computation rules*) saying how the eliminator acts on canonical elements. We treat types themselves as being elements of special types called *universes*, representing types of types. To avoid circularity issues, we have a hierarchy of universes, $\mathcal{U}_0, \mathcal{U}_1, \ldots$. Universes are cumulative (if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$) but are not transitive (so $a : A$ and $A : \mathcal{U}_i$ does not mean $a : \mathcal{U}_i$), and we have $\mathcal{U}_i : \mathcal{U}_{i+1}$ for each $\mathcal{U}_i$. We will work in an ambiguous universe $\mathcal{U}$, only indexing the universe when we are discussing size issues. As such, we will write $A : \mathcal{U}$ to mean that A is a type. Type families indexed by a type A will be treated as functions $B : A \to \mathcal{U}$.

We will often use types as statements, in which case we mean to say that they have an element. For example, we state many theorems by saying something of the form "We have P.", which means "There is some (unspecified) $p : P$". In fact, all statements and constructions in our system reduce to the construction of some element $p : P$, and we prove all of them by giving an explicit element. We discuss this in more depth in Section 2.2.

The first type former of interest is that of *dependent products*: For any type A and type family $B : A \to \mathcal{U}$, we have a *dependent product* type, or *type of dependent functions*, $\Pi(a : A), B(a) : \mathcal{U}$. Elements are introduced via lambda abstraction: if for any $x : A$, we have $t : B(x)$ then there is an element $\lambda x.\, t : \Pi(a : A), B(a)$. We will sometimes write definitions by giving arguments to the function, so $f(x) \stackrel{\mathsf{def}}{=} y$ means $f \stackrel{\mathsf{def}}{=} \lambda x.y$. The elimination rule is function application, and the computation rule says that $(\lambda x.\, t)(a) \equiv t[x/a]$. The type $A \to B$ of *functions* from A to B is the product over the constant family $\lambda(x : A).\, B$.

We treat a function of two arguments (of type A and type B) as a *curried* function of type $A \to (B \to C)$, or in the dependent case, if $B : A \to \mathcal{U}$ and $C : \Pi(a : A), (B(a) \to \mathcal{U})$, as a function $\Pi(a : A), (\Pi(b : B(a)), C(a)(b))$. When we have $f : \Pi(a : A), (\Pi(b : B(a)), C(a)(b))$ we may write $f(a)(b)$, or $f(a, b)$ or $f_a(b)$ for the application of f to a and b, depending on focus and readability. To simplify notation, we treat quantifiers (such as Π) as binding as far right as possible, and $\to$ as right-associative, so that we can write the types above as $A \to B \to C$ and $\Pi(a : A), \Pi(b : B(b)), C(a)(b)$. The most basic example of a function is the identity function,

$$
\begin{aligned}
&\mathsf{id} \;\; : \Pi(A : \mathcal{U}), (A \to A), \\
&\mathsf{id}_A \equiv \lambda x.x.
\end{aligned}
$$

The computation rules are sometimes know as η *rules* , and the η rule for functions warrants some discussion. Consider the non-dependent composition operator

$$
- \circ - : (B \to C) \to (A \to B) \to (A \to C)
$$

defined by

$$g \circ f \stackrel{\mathsf{def}}{=} \lambda x.g(f(x))$$

Then the η rules says that $h \circ (g \circ f) \equiv (h \circ g) \circ f$, as we have

$$h \circ (g \circ f) \stackrel{\mathsf{def}}{=} \lambda x.h(g \circ f(x)) \equiv \lambda x.h((\lambda x.g(f(x)))(x)) \equiv \lambda x.h(g(f(x))) \equiv \lambda x.(h \circ g)(fx) \equiv (h \circ g) \circ f.$$

In fact, in general, if two functions $f, g : A \to B$ are such that $f(x) \equiv g(x)$ for every $x : A$, then we have $f \equiv g$. However, judgmental equality should be thought of as part of the meta-system, and so we cannot express this fact in our system. We will shortly introduce a type-level or *propositional* version of equality, and the corresponding statement (known as *function extensionality*) for propositional equality can be expressed internally, but does not hold in pure MLTT. We discuss function extensionality in more detail in Section 2.1.

Most of the rest of our type formers will be *inductive types*: the elimination rules are *induction principles* saying that a (dependent) map out of the type is defined uniquely by specifying its behavior on the canonical elements. We have,

- An *empty type*, $\emptyset : \mathcal{U}$ with no introduction rule, so that for any $C : \mathcal{U}$ we have a unique function $! : \emptyset \to C$; more generally, if $C : \emptyset \to \mathcal{U}$ then we have a unique dependent function $! : \Pi(z : \emptyset), C(z)$. We interpret statements of the form "Not P" as meaning $P \to \emptyset$.
- A *unit type*, $1 : \mathcal{U}$ with a single canonical element $\star : 1$.
- A *boolean type*, $2 : \mathcal{U}$ with canonical elements $0, 1 : 2$.
- A type $\mathbb{N} : \mathcal{U}$ of *natural numbers* with canonical elements $0 : \mathbb{N}$ and $\mathsf{succ}(n) : \mathbb{N}$ for each $n : \mathbb{N}$. As usual, we will often write $n + 1$ for $\mathsf{succ}(n)$.
- For each $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ a *sum type*, or type of *dependent pairs*, $\Sigma(a : A), B(a)$ with canonical elements $(a, b) : \Sigma(a : A), B(a)$ for each $a : A$ and $b : B(a)$. The *cartesian product* $A \times B$ of A and B is taken to be the sum over the constant family $\lambda(x : A).\, B$
- For $A, B : \mathcal{U}$, we have a *coproduct* or *disjoint union* type $A + B$ with canonical elements $\mathsf{inl}(a) : A + B$ for each $a : A$ and $\mathsf{inr}(b) : A + B$ for each $b : B$.

A member of $\Sigma(x : A), B(x)$ is thought of as a pair, and this is expressed by the elimination rules for Σ. In particular, we have

$$\begin{aligned}
&\mathsf{pr}_0 && : \big(\Sigma(x : A), B(x)\big) \to A \\
&\mathsf{pr}_0(a, b) && \stackrel{\mathsf{def}}{=} a \\
&\mathsf{pr}_1 && : \Pi(p : \Sigma(x : A), B(x)), B(\mathsf{pr}_0(p)) \\
&\mathsf{pr}_1(a, b) && \stackrel{\mathsf{def}}{=} b.
\end{aligned}$$

These two functions are completely specified by giving their value on constructors. We will sometimes abuse notation writing something of the form $\lambda(a, b).t(a, b)$ to mean the function defined by induction with value $t(a, b)$ on the pair (a, b). As with Π, we take Σ to bind as far right as can be made sense of. We also treat $(-, -)$ as right associative, so (a, b, c, d) means $(a, (b, (c, d)))$ and is an element of the type

$$\Sigma(a : A), \Sigma(b : B(a)), \Sigma(c : C(a, b)), D(a, b, c),$$

while $((a, b), c)$ is an element of the type

$$\Sigma(p : \Sigma(x : A), B(x)), C(\mathsf{pr}_0(p), \mathsf{pr}_1(p)).$$

or less formally, of the type

$$\Sigma((x, y) : \Sigma(x : A), B(x)), C(x, y),$$

However, as expected, pairing is associative (up to equivalence), and we will be sloppy with our pairing notation when doing so aids readability.

Additionally, for each $A : \mathcal{U}$, we have the inductive family $\mathsf{Id}_A : A \to A \to \mathcal{U}$ of *identity types*. We will usually write $a = b$ or $a =_A b$ for $\mathsf{Id}_A(a, b)$, and we call elements of $a = b$ *paths* or *equalities*. The introduction rule gives $\mathsf{refl}_a : a = a$ and the elimination rule, called *path induction*, is as follows:

Given

- $C : \Pi(a, b : A), a = b \to \mathcal{U}$, and
- $c : \Pi(a : A), C(a, a, \mathsf{refl}_a)$;

we have an inhabitant

$$f : \Pi(a, b : A), \Pi(p : a = b), C(a, b, p).$$

such that

$$f(\mathsf{refl}_a) \equiv c(a).$$

The element expressing the elimination rule is traditionally called J:

$$J_{A,C} : \Big(\Pi(a : A), C(a, a, \mathsf{refl})\Big) \to \Pi(a, b : A), \Pi(p : a = b), C(a, b, p).$$

Note that here we have used a *judgemental* equality, which is also assumed for the inductive types given above. However, as we work informally, we will be vague about the difference between judgemental equality and the type-level equalities arising from identity types. In short, we only work up to type-level equality.

There is also a *based path induction* principle which says that for fixed $a : A$ and given

- $C : \Pi(b : A), a = b \to \mathcal{U}$, with
- $c : C(a, \mathsf{refl}_a)$;

we have an inhabitant

$$f : \Pi(b : A), \Pi(p : a = b), C(b, p)$$

such that

$$f(\mathsf{refl}_a) \equiv c(a).$$

In practice, we usually present proofs by path induction by assuming p is refl. We prove the following lemma first in a more rigorous style using J, and then in the more intuitive style we use subsequently.

Lemma 1.1. *Fix a type $A : \mathcal{U}$.*

(i) For any $x, y : A$ and $p : x = y$, there is $p^{-1} : y = x$.

(ii) For any $x, y, z : A$ any $p : x = y$ and $q : y = z$, we have $p \cdot q : x = z$ such that each of the following equations hold for all $p : x = y$:

$$p \cdot \mathsf{refl}_y = p,$$

$$\mathsf{refl}_x \cdot q = q,$$

$$p \cdot p^{-1} = \mathsf{refl}_x,$$

$$p^{-1} \cdot p = \mathsf{refl}_y.$$

Proof. (Using J.)

(i) Define $C : \Pi(x, y : A), (x = y) \to \mathcal{U}$ by

$$C(x, y, p) \stackrel{\mathsf{def}}{=} y = x,$$

and note that for any $a : A$ we have $C(a, a, \mathsf{refl}_a) \equiv (a = a)$, so then define $c : \Pi(a : A), C(a, a, \mathsf{refl}_a)$ by $c(a) = \mathsf{refl}_a$. Then letting $f \stackrel{\mathsf{def}}{=} J_{A,C}(c)$ we have

$$f : \Pi(x, y : A), (x = y) \to (x = y),$$

satisfying the equation

$$f(a, a, \mathsf{refl}_a) \equiv \mathsf{refl}_a.$$

Now define $p^{-1} = f(p)$.

(ii) Define $C : \Pi(x, y : A), (x = y) \to \mathcal{U}$ by

$$C(x, y, p) \stackrel{\mathsf{def}}{=} \Pi(z : Z), (y = z) \to (x = z).$$

Note that for any $a : A$ we have $C(a, a, \mathsf{refl}_a) \equiv \Pi(z : Z), (a = z) \to (a = z)$, so define $c : \Pi(a : A), C(a, a, \mathsf{refl}_a)$ by $c(a) = \lambda z, p.p$. Let $f \overset{\mathsf{def}}{=} J(c)$ and for $p : x = y$ and $q : y = z$, define $p \cdot q \overset{\mathsf{def}}{=} f(x, y, p)(z, q)$.

That $\mathsf{refl}_x \cdot q = q$ follows by definition. For the other two equations, we need to again use the elimination principle.

We need to check the equations. Define $C, C', D, D' : \Pi(x, y : A), (x = y) \to \mathcal{U}$ as

$$
\begin{aligned}
C(x, y, p) &\overset{\mathsf{def}}{=} p \cdot \mathsf{refl}_y = p \\
C'(x, y, p) &\overset{\mathsf{def}}{=} \mathsf{refl}_x \cdot p = p \\
D(x, y, p) &\overset{\mathsf{def}}{=} p \cdot p^{-1} = \mathsf{refl}_x \\
D'(x, y, p) &\overset{\mathsf{def}}{=} p^{-1} \cdot p = \mathsf{refl}_y
\end{aligned}
$$

Then we have

$$
\begin{aligned}
c &: \Pi(a : A), C(a, a, \mathsf{refl}_a) \\
c_a &\overset{\mathsf{def}}{=} \mathsf{refl}_{\mathsf{refl}_a}, \\
c' &: \Pi(a : A), C'(a, a, \mathsf{refl}_{\mathsf{refl}_a}) \\
c'_a &\overset{\mathsf{def}}{=} \mathsf{refl}_{\mathsf{refl}_a} \\
d &: \Pi(a : A), D(a, a, \mathsf{refl}_a) \\
d_a &\overset{\mathsf{def}}{=} \mathsf{refl}_{\mathsf{refl}_a} \\
d'_a &: \Pi(a : A), D'(a, a, \mathsf{refl}_a) \\
d'_a &\overset{\mathsf{def}}{=} \mathsf{refl}_{\mathsf{refl}_a}
\end{aligned}
$$

Then we have

$$\begin{aligned}
J_{A,C}(c) &: \Pi(x,y:A), \Pi(p:x=y), p \cdot \mathsf{refl}_y = p \\
J_{A,C'}(c') &: \Pi(x,y:A), \Pi(p:x=y), \mathsf{refl}_x \cdot p = p \\
J_{A,D}(d) &: \Pi(x,y:A), \Pi(p:x=y), p \cdot p^{-1} = \mathsf{refl}_x \\
J_{A,D'}(d') &: \Pi(x,y:A), \Pi(p:x=y), p^{-1} \cdot p = \mathsf{refl}_y
\end{aligned}$$ □

Proof. (By reduction to the case that x and y are the same and p is refl.)

(i) Let $x \equiv y$ and p be refl. Then define $\mathsf{refl}^{-1} \stackrel{\mathsf{def}}{=} \mathsf{refl}$.

(ii) Let $x \equiv y$ and p be refl. Then define $\mathsf{refl} \cdot q \stackrel{\mathsf{def}}{=} q$, so that we have $\mathsf{refl} \cdot q \equiv q$.

We also have $\mathsf{refl} \cdot \mathsf{refl}^{-1} \equiv \mathsf{refl} \cdot \mathsf{refl} \equiv \mathsf{refl}$. Now suppose q is refl, so that we have a path $\mathsf{refl}_x \cdot \mathsf{refl}_y \equiv \mathsf{refl}_x$ and so for any p, we have $p \cdot \mathsf{refl}_y = p$. □

1.2 Bishop-style mathematics and the Curry-Howard interpretation

Type theory arose from an intuitionistic perspective, wherein mathematical constructions are at least as basic as logical operations, in contrast to a first-order theory such as ZFC, which builds mathematics on top of logical deduction. In other words, to express logic in our language, logical notions must be encoded using types. This is traditionally done via the *Curry-Howard* interpretation of logic: we interpret propositions as types, and take a proof of the proposition P to be the construction of an element $p : P$. Under this interpretation, the logical operations are

$$\begin{aligned}
P \wedge Q &\equiv P \times Q \\
P \vee Q &\equiv P + Q \\
P \Rightarrow Q &\equiv P \to Q \\
\neg P &\equiv P \to \emptyset \\
\forall(a:A), P(a) &\equiv \Pi(a:A), P(a) \\
\exists(a:A), P(a) &\equiv \Sigma(a:A), P(a).
\end{aligned}$$

For example, the statement

$$\forall(x \in \mathbb{N}).x \neq 0 \to \exists(y \in \mathbb{N}).x = \mathsf{succ}(y)$$

becomes the type

$$\Pi(x : \mathbb{N}), \big(x = 0 \to \emptyset\big) \to \Sigma(y : \mathbb{N}), x = \mathsf{succ}(y),$$

and this type has an element, by induction on x:

In the base case, we want to see $(0 = 0 \to \emptyset) \to \Sigma(y : \mathbb{N}), x = \mathsf{succ}(y)$. Let $f : 0 = 0 \to \emptyset$; as we have $\mathsf{refl} : 0 = 0$, we then get $f(\mathsf{refl}) : \emptyset$, and so by the elimination principle for $\emptyset$ (i.e., the unique function from $\emptyset$ into any other type), we have $!(f(\mathsf{refl})) : \Sigma(y : \mathbb{N}), x = \mathsf{succ}(y)$.

For the inductive step, we know x is of the form $\mathsf{succ}(n)$. Then

$$\lambda f.(n, \mathsf{refl}_{\mathsf{succ}(n)}) : \big(x = 0 \to \emptyset\big) \to \Sigma(y : \mathbb{N}), x = \mathsf{succ}(y).$$

Using the induction principle then gives us an element of the desired type—a proof of the fact that every non-zero natural number is a successor.

The Curry-Howard interpretation allows the interpretation of logical concepts which matches the constructivist explanation of the logical connectives. However, it quickly becomes apparent that many *extensional concepts* [35]—concepts specifying the behavior of equality—should be added to MLTT in order to work with MLTT effectively. Four noteworthy extensional concepts are *function extensionality*, saying that pointwise equal functions are equal; proposition extensionality, saying that logically equivalent propositions are equal; proof irrelevance, saying that any two proofs of the same proposition are equal; and quotient types, identifying related elements of a type. The traditional way to resolve these difficulties is to equip types with equivalence relations on demand, instead of using the identity types introduced earlier. This matches Bishop's view that to give a set is to give not only its members, but also what we must show to show two members to be equal. A type equipped with an equivalence relation is called a *setoid*. Given two setoids $(A, \approx)$ and $(B, \approx)$, there is no reason to expect an arbitrary $f : A \to B$ to respect the equivalence relation. It is typical to therefore call elements of $A \to B$ *operations* from A to B, and call an operation $f : A \to B$ a *function* when for all $a, a' : A$ if $a \approx a'$ the $f(a) \approx f(a')$.

When working with setoids, we can force function extensionality to hold by equipping the type of functions with the equivalence relation of pointwise equality. Similarly, we may introduce quotients by simply adjusting the equivalence relation on the setoid: if $(A, \approx)$ is a setoid, and E as an equivalence relation on A which is coarser than $\approx$, we can set A/E to be the setoid (A, E). More subtly, we may consider a class of proof-irrelevant setoids: say that $(A, \approx)$ is *proof-irrelevant* when $\approx$ is the chaotic relation—that is, when $a \approx a'$ for any $a, a' : A$.

We will say more about setoids in Chapter 6, after we examine the extensional concepts we are interested in from the univalent perspective. For now, it is worth remarking that it is some work to equip Σ and Π types with setoids (a notion of substitution is required for this), and that it is not clear whether there is a reasonable way to make the universe into a setoid. As the universe is central to our approach to partial functions in Chapter 5, the setoid approach will not work for our purposes.

1.3 Informal mathematics and constructive taboos

The logic we get in a Bishop-style mathematics and the logic we get in the univalent mathematics we will use are both constructive. As a result, we do not have the law of the excluded middle. A consequence is that the possibility of an independent or *constructively undecided* proposition is always in the air. Discussion around these principles can be misleading at times. Consider the following two sentences

- "The type of natural numbers is discrete."
- "The type of real numbers cannot be discrete."

While we do not yet know the definition of discreteness, it is enough to know that if LEM holds, then all sets are discrete. Regardless, the first is a theorem in our system (as a consequence of Theorem 1.13) and the second is an *informal metatheorem*. We could make this into an honest metatheorem by (e.g.) proving that there is no proof that the real numbers are discrete, but we want to avoid a deep detour into metamathematics. Such informal statements are often justified in constructive math by appeal to *constructive taboos*: statements which we expect not to be provable in a constructive system. Statements of the second form are explanations of results of the form $P \to Q$, where Q is a constructive taboo. In particular, if the type of real numbers has decidable equality, then we can derive Bishop's *limited principle of omniscience*, which we can write in a logical style as

$$\mathrm{LPO} \overset{\mathsf{def}}{=} \forall(\alpha : \mathbb{N} \to 2), \big(\exists(n : \mathbb{N}), \alpha(n) = 1)\big) \vee \neg\big(\exists(n : \mathbb{N}), \alpha(n) = 1\big).$$

As the limited principle of omniscience says that we can examine infinitely many cases, it does not meet Bishop's requirement that we are describing "finitely performable abstract operations", and so a system which allows us to prove LPO is by its nature non-constructive. Then we cannot expect to have the type of real numbers to be discrete in a constructive system.

Some constructive systems arise from considerations in analysis or computability theory [13], and often contain axioms that are incompatible with the law of the excluded middle. In this thesis, such principles will also be called taboos. Below we list several taboos that we will make mention of throughout. The list contains an informal statement of each, a brief explanation of the reasons the principle might be considered, and a formal logical statement, and the same statement in the Curry-Howard interpretation.

Limited principle of omniscience (LPO) Every binary sequence either takes the value 1, or doesn't take the value 1. First introduced by Bishop [9] as a principle which should not be constructively valid.

$$\forall(\alpha : \mathbb{N} \to 2), \big(\exists(n : \mathbb{N}), \alpha(n) = 1)\big) \vee \neg\big(\exists(n : \mathbb{N}), \alpha(n) = 1\big).$$

$$\Pi(\alpha : \mathbb{N} \to 2), \big(\Sigma(n : \mathbb{N}), \alpha(n) = 1)\big) + \neg\big(\Sigma(n : \mathbb{N}), \alpha(n) = 1\big).$$

Weak limited principle of omniscience (WLPO) Every binary sequence is either constantly 0 or not constantly 0. LPO weakened by replacing Q by $\neg\neg Q$, where Q is $\exists(n : \mathbb{N}), \alpha(n) = 1$. Of interest when considering computability, as here it should fail (see Section 8.2).

$$\forall(\alpha : \mathbb{N} \to 2), \big(\forall(n : \mathbb{N}), \alpha(n) = 0)\big) \vee \neg\big(\forall(n : \mathbb{N}), \alpha(n) = 0\big).$$

$$\Pi(\alpha : \mathbb{N} \to 2), \big(\Pi(n : \mathbb{N}), \alpha(n) = 0)\big) \vee \neg\big(\Pi(n : \mathbb{N}), \alpha(n) = 0\big).$$

Markov's Principle (MP) If a binary sequence is not constantly 0, then it takes the value 1. Introduced by Markov [56] based on considerations in recursion theory.

$$\forall(\alpha : \mathbb{N} \to 2), \big(\neg\forall(n : \mathbb{N}), \alpha(n) = 0)\big) \to \big(\exists(n : \mathbb{N}), \alpha(n) = 1\big).$$

$$\Pi(\alpha : \mathbb{N} \to 2), \big(\neg\Pi(n : \mathbb{N}), \alpha(n) = 0)\big) \to \big(\Sigma(n : \mathbb{N}), \alpha(n) = 1\big).$$

Kripke's Schema (KS) For each proposition P, there is a binary sequence such that P holds iff this sequence takes the value 1. Introduced to formalize Brouwer's concept of the *creative subject*. See Section 4.10 of [85] for some history. Allowing quantification over propositions, this is written

$$\forall(P : \mathsf{Prop}), \exists(\alpha : \mathbb{N} \to 2), \big(P \leftrightarrow (\exists(n : \mathbb{N}), \alpha(n) = 1)\big).$$

Under Curry-Howard, we take Prop to be $\mathcal{U}$:

$$\Pi(P : \mathcal{U}), \Sigma(\alpha : \mathbb{N} \to 2), \big(P \leftrightarrow (\Sigma(n : \mathbb{N}), \alpha(n) = 1)\big).$$

Church's Thesis (CT) Every total function $\mathbb{N} \to \mathbb{N}$ is computable. Arises from computability considerations, as all constructively definable functions ought to be computable.

$$\forall(f : \mathbb{N} \to \mathbb{N}), \exists(e : \mathbb{N}), \forall(n : \mathbb{N}), f(n) = \{e\}(n).$$

$$\Pi(f : \mathbb{N} \to \mathbb{N}), \Sigma(e : \mathbb{N}), \Pi(n : \mathbb{N}), f(n) = \{e\}(n).$$

Here $\{e\}$ is the function coded by e.

The Curry-Howard statements of both Kripke's Schema and Church's thesis are false, even though Kripke's Schema follows from LEM, and there are constructive settings in which Church's thesis holds [43]. We will introduce a more refined interpretation of logical principles in Section 2.2. Using this interpretation, we consider a more reasonable version of Church's thesis in Section 9.2. A version of Kripke's Schema which seems to better capture the intent of the schema can be given using ideas from Chapter 5.

1.4 Topos logic

It is possible to interpret Martin-Löf type theory in a topos $\mathcal{E}$, following Hoffman's modification of Seely's interpretation in any locally cartesian closed category [77, 36]. Briefly, we take the objects of $\mathcal{E}$ to represent contexts of free variables, and a type in context Γ is a map into Γ (an object of $\mathcal{E}/\Gamma$), while an element of that type is a section of this map. Any map $f : \Delta \to \Gamma$ gives rise to a function $f^* : \mathcal{E}/\Gamma \to \mathcal{E}/\Delta$ by pullback. The left-adjoint of this is used to interpret Σ, while the right adjoint is used to interpret Π.

The interpretation outlined above gives a way to interpret informal mathematics in a topos, by appealing to the Curry-Howard interpretation in the type theory associated to the topos. However, it is more common to interpret informal mathematics in a topos by appealing to the *Mitchell-Benabou* language of a topos. The Mitchell-Benabou language for a topos $\mathcal{E}$ is a higher-order logic which arises by considering the subobject classifier Ω of $\mathcal{E}$. We roughly follow the presentation in [55]. A more detailed account can be found there, as well as in [51] and [59].

For each object X of $\mathcal{E}$, there is a *type* X, and each type has a set of *terms*. Each term in the language additionally has free variables of fixed types. A term of type B with free variables of types $A_1, \dots, A_n$ will be written

$$x_1 : A_1 \dots, x_n : A_n \vdash t : B.$$

A term $x_1 : A_1 \dots, x_n : A_n \vdash t : B$ will be interpret by a morphism $A_1 \times \cdots \times A_n \to B$. The basic data of the LCCC structure of $\mathcal{E}$ give terms of the language corresponding to the morphisms given by the data. We omit these, except for those given by exponentiation which are illustrative: Given a term $x : U \vdash \sigma : X$ interpreted by $\chi : U \to X$ and a term $v : V \vdash f : Y^X$ whose interpretation is given by $\beta : V \to Y^X$, there is a term $x : U, y : V \vdash f(v) : Y$ whose interpretation is given by

$$V \times U \xrightarrow{\langle \chi, \beta \rangle} X \times Y^X \xrightarrow{e} Y,$$

where e is the evaluation map. Given a term $x : X, u : U \vdash Z$ which is interpreted by θ, there is a term $x : X \vdash \lambda x.\theta : Z^U$ whose interpretation is the transpose of θ.

The terms of interest to us here are those whose type is Ω, which we call *formulas*. As there are maps $\wedge, \vee, \Rightarrow : \Omega \to \Omega$ and a map $\neg : \Omega \to \Omega$ corresponding to the logical connectives, we can interpret the logical connectives on formulas. What remains is interpreting equality and quantifiers. As the diagonal $\Delta : X \to X \times X$ is an embedding, it has a characteristic function $\delta : X \times X \to \Omega$, and so given terms $u : U \vdash \sigma : X$ and $v : V \vdash \tau : X$, we have a formula $u : U, v : V \vdash (\sigma = \tau) : \Omega$ interpreted by composing $\langle \sigma, \tau \rangle$ with δ.

The quantifiers are slightly more work. The formula $x : X, y : Y \vdash \varphi : \Omega$ is interpreted as a map $X \times Y \to \Omega$, which corresponds to an arrow $\lambda x.\varphi : Y \to \Omega^X$. The unique map $X : X \to 1$ gives rise to a map $1^\Omega \to X^\Omega$, which corresponds to the functor $(!_X)^* : \mathcal{E}/1 \to \mathcal{E}/X$, and we know that $1^\Omega = \Omega$. The left and right adjoints give us maps $\forall_X, \exists_X : \Omega \to \Omega^X$. Then we have maps $\forall_X \circ \lambda x.\varphi : Y \to \Omega$ and $\exists_X \circ \lambda.x\varphi : Y \to \Omega$, which we take as the interpretations of $\forall(x : X), \varphi(x, y)$ and $\exists(x : X), \varphi(x, y)$.

As a formula is treated as a map $X \to \Omega$, any formula φ corresponds to a subobject of X, and we can write $\{x \mid \varphi(x)\}$ for the domain of this subtype. In short, we have interpreted equality, the logical connectives, quantifiers, and a form of comprehension. Then we can use this to interpret a great deal of mathematical work. Moreover, the more structure our topos has, the more mathematics we can interpret: for example, if the topos contains a natural numbers object, we can interpret the natural numbers, and arithmetic in the topos.

The interpretation of MLTT and the Mitchell-Benabou language of a topos are not entirely independent: give an object X of $\mathcal{E}$, there is the formula (with no free variables) $\|X\| \stackrel{\text{def}}{=} \exists x.\top$ which we can call the *truncation* of X. If $P : X \to \Omega$, we can form not only the formula $\exists(x : X), P(x)$, but also the type $\Sigma(p : P), (X)$, and in fact we have an isomorphism between the truncation $\|\Sigma(p : P), (X)\|$ and $\exists(x : X), P(x)$. In other words, in a topos, existential quantification can be defined from Σ and truncation. In univalent mathematics, we will do the same thing: since we are working in a type theory, we have Σ, and we will then define a *truncation operator* (See Section 2.2), which we will use to define the existential quantifier. This gives way to a third, more nuanced interpretation of mathematical ideas where we are allowed to mix higher-order logic and type theory in a fluid way. We call mathematics done in this style *univalent mathematics*.

1.5 Univalent foundations

In Lemma 1.1, we proved that we can compose and invert the elements of identity types. When we include the reflexivity path refl, we can see this result as expressing that identity is an equivalence relation; as the elimination principle can be seen as saying that identity is the smallest reflexive relation, this also makes identity into the smallest equivalence relation. However, $x = y$ is not simply a proposition, but a *type*—to assert $x = y$ is not simply to give a truth value, but to give the data of *which* inhabitant of $x = y$ we have. This is to say, identity is not simply property in Martin-Löf type theory, but *structure*.

Univalent mathematics arises from the observation that the structure of identity types $x =_A y$ is part of the structure of the type A. For example, the fact (which we prove shortly) that $\Pi(n, m : \mathbb{N}), (n = m) + \neg(n = m)$ is part of the description of the structure of $\mathbb{N}$. We can then stratify the types into levels based on how deep the identity structure on a type goes. These levels, called *homotopy-levels* capture a notion of dimension. Traditionally, these dimensions are indexed from -2, following the indexing in homotopy theory; we use $\mathbb{N}_{-2}$ for the type of integers at least -2. The most important are the first four levels, those of *contractible types*, *propositions*, *sets* and *groupoids*. In order, these are types with exactly one element, types with at most one element, types whose identity types are propositions, and types whose identity types are sets. More explicitly:

Definition 1.2. We define the operations $\mathsf{isProp} : \mathcal{U} \to \mathcal{U}$, $\mathsf{isContr} : \mathcal{U} \to \mathcal{U}$, $\mathsf{isSet} : \mathcal{U} \to \mathcal{U}$ and $\mathsf{isGroupoid} : \mathcal{U} \to \mathcal{U}$ by

$$\begin{aligned}
\mathsf{isContr}(X) &\stackrel{\mathsf{def}}{=} \Sigma(a : X), \Pi(x : X), (a = x), \\
\mathsf{isProp}(X) &\stackrel{\mathsf{def}}{=} \Pi(x, y : X), x = y, \\
\mathsf{isSet}(X) &\stackrel{\mathsf{def}}{=} \Pi(x, y : X), \mathsf{isProp}(x = y), \\
\mathsf{isGroupoid}(X) &\stackrel{\mathsf{def}}{=} \Pi(x, y : X), \mathsf{isSet}(x = y).
\end{aligned}$$

We call a type X a *proposition* or a *subsingleton* if $\mathsf{isProp}(X)$, *contractible* or a *singleton* if $\mathsf{isContr}(X)$ a *set* if $\mathsf{isSet}(X)$ and a *groupoid* if $\mathsf{isGroupoid}(X)$. We call a type family $P : X \to \mathcal{U}$ a *predicate* when $\Pi(x : X), \mathsf{isProp}(P(x))$. We also have the types $\mathsf{Prop}, \mathsf{Set} : \mathcal{U}_1$,

$$\mathsf{Prop} \stackrel{\mathsf{def}}{=} \Sigma(P : \mathcal{U}), \mathsf{isProp}(P).$$

$$\mathsf{Set} \stackrel{\mathsf{def}}{=} \Sigma(X : \mathcal{U}), \mathsf{isSet}(X).$$

For any $n : \mathbb{N}_{-2}$, the *homotopy n-types*, are given by

$$\begin{aligned}
\mathsf{is\text{-}}(-2)\mathsf{\text{-}type}(X) &\stackrel{\mathsf{def}}{=} \mathsf{isContr}(X), \\
\mathsf{is\text{-}}(n+1)\mathsf{\text{-}type}(X) &\stackrel{\mathsf{def}}{=} \Pi(x, y : X), \mathsf{is\text{-}}n\mathsf{\text{-}type}(x = y).
\end{aligned}$$

We will be working primarily at the level of sets and propositions, but as far as possible, we will state our results for general types. We will also make use of examples and counterexamples which have higher levels. The question of whether a type is a set is not a size issue, in the way that the class of all sets is too large to be a set, but rather, it is an issue of depth of structure. Size issues are handled by universe levels. For example, we gave the type Set as a type in $\mathcal{U}_1$. More explicitly, we have for each i, a type $\mathsf{Set}_i : \mathcal{U}_{i+1}$ with $\mathsf{Set}_i \stackrel{\mathsf{def}}{=} \Pi(X : \mathcal{U}_i), \mathsf{isSet}(X)$.

Note that the definition of $\mathsf{isProp}(P)$ and $\mathsf{is\text{-}}(-1)\mathsf{\text{-}type}(P)$ are not quite the same. We will show shortly that these two notions are in fact equivalent, and so also that sets are 0-types and groupoids are 1-types. However, we do not yet have the machinery to talk about equivalence properly.

In the meantime, we use a notion of *logical equivalence*: Define

$$A \Leftrightarrow B \stackrel{\mathsf{def}}{=} (A \to B) \times (B \to A)$$

and say that A and B are *logically equivalent* if $A \Leftrightarrow B$. The name is justified by Lemma 1.7, which says that when two propositions are logically equivalent, they are in fact equivalent (in the sense defined in the next section).

The canonical examples of propositions are $\emptyset$ and 1.

Theorem 1.3. *The empty type $\emptyset$ and unit type 1 are propositions. The unit type is contractible, while the empty type is not.*

Proof. The elimination principle for $\emptyset$ says directly that $\Pi(x, y : \emptyset), x = y$. Moreover, the first projection $\Big(\Sigma(x : \emptyset), \Pi(y : \emptyset), x = y\Big) \to \emptyset$ gives a map $\mathsf{isContr}(\emptyset) \to \emptyset$.

As we have $\mathsf{refl} : \star = \star$, by the elimination principle for the unit type we have some witness $w : \Pi(x : 1), \star = x$, and so $(\star, w) : \mathsf{isContr}(1)$. However, again by applying the elimination principle for 1 to w, we get $\Pi(x, y : 1), y = x$, by symmetry we have $\mathsf{isProp}(1)$. $\square$

The type $\mathbb{N}$ is a set which is not a proposition. The proof relies on the notion of equivalence, so we give it in the next section.

When defining an algebraic structure, for example, a group, it is most natural to work with underlying types which are *sets*, since we want the group equations to be property, with the structure given only by the operations. Regardless, we may define $\mathsf{GrpStruct} : \mathcal{U} \to \mathcal{U}$ by

$$\begin{aligned}\mathsf{GrpStruct}(X) \overset{\mathsf{def}}{=} \Sigma(- \cdot - : X \to X \to X), \Sigma((-)^{-1} : X \to X), \Sigma(e : X),\\ \Pi(x, y, z : X), \big(x \cdot (y \cdot z) = (x \cdot y) \cdot z\big)\\ \times \Pi(x : X), \big((x \cdot e = x) \times (e \cdot x = x) \times (x \cdot x^{-1} = e) \times (x^{-1} \cdot x = e)\big),\end{aligned}$$

The first line says that we have the operations $- \cdot -$ and $(-)^{-1}$, as well as an identity e; the second line ensures associativity, and the third the identity and inverse rules. This is a direct formalization of the classical definition:

> A *group structure* on X consists of a multiplication operation $- \cdot - : X^2 \to X$, an inverse operation $(-)^{-1} : X \to X$ and an element $e \in X$ satisfying the equations
>
> $$\begin{aligned}(\forall x, y, z \in X) \quad & x \cdot (y \cdot z) = (x \cdot y) \cdot z\\ (\forall x \in X) \quad & x \cdot e = e \cdot x = x\\ (\forall x \in X) \quad & x \cdot x^{-1} = e = x^{-1} \cdot x\end{aligned}$$

All of the operations on types are such formalizations. For example, a type P is a proposition if for all $x, y : P$, we have that $x = y$.

In the case that X is a set, the required equations can be shown to be propositions—that is, we can show that the equations are property. Then we can define $\mathsf{Grp} : \mathcal{U}_1$:

$$\mathsf{Grp} \overset{\mathsf{def}}{=} \Pi(X : \mathcal{U}), \mathsf{isSet}(X) \times \mathsf{GrpStruct}(X).$$

If we assume Voevodsky's univalence axiom (see Section 2.4), we can show that equality in Grp is exactly isomorphism [24]. As we may have many non-trivial isomorphisms between two groups, we know that we cannot have that Grp is a set. In fact in this case we can show that Grp is a groupoid.

1.6 Structure versus Property

Before moving on, we should further discuss structures and properties. In a traditional development of semi-formal mathematics, we establish the truth (or non-truth) of statements. For example, one may express the first isomorphism theorem in such a development as

For any surjective homomorphism $\varphi : G \to H$, there is an isomorphism $G/\ker\varphi \simeq H$.

This is a perfectly honest theorem, but it asserts only a simple fact: it tells us that we can identify $G/\ker\varphi$ and H. This fact gives us no information about what happens to a given $g \in G$ under this identification, despite the fact that when using the isomorphism theorem, we are often interested in a particular isomorphism between $G/\ker\varphi$ and H. A better version of the theorem is given by

For any surjective homomorphism $\varphi : G \to H$, the natural function $\nu : G/\ker\varphi \to H$ defined by $\nu([g]) = \varphi(g)$ is an isomorphism.

This version contains *structure*: we not only know that $G/\ker\varphi$ can be identified with H, but we know which map gives this identification, and how it behaves. We can think of an isomorphism as a structured identification between groups, so that in rephrasing the theorem, we have moved from a simple statement of fact to a description of structure. On the other hand the property that f is an isomorphism between groups is something we usually do not wish to analyze for deeper structure: the fact that f is an isomorphism is all we are interested in.

In a formalization based on first order (or higher order) logic, we manipulate facts; structure is not easily expressed by the formal manipulations. If we want the formal framework underlying our mathematics to be able to manipulate structure directly, then another sort of system is needed. The obvious choice is to turn to some variant of MLTT. However, logic is a part of mathematical reasoning, and MLTT does not natively have a notion of truth value, so some convention on how to interpret logical operations must be taken. Propositions can be represented by types, and then we say that a proposition P is true if there is some inhabitant $p : P$. The traditional convention is that of Section 1.2. Namely, view *all* types as propositions and use a Curry-Howard interpretation of logic. But then we lose our simple facts: even $3 + 4 = 7$ is (a priori) a type with structure which can be analyzed.

Our example above suggests that mathematics is concerned with both facts and structures. Traditional foundations often only directly express the former, while type theoretic foundations often only directly express the later. Univalent definitions allow us to talk about both, and to relate them. Arbitrary types represent structure or data. Propositions represent simple facts—the definition tells us that the structure of the type is trivial. In principle, we can give two versions of a notion: a propositional version (expressing a simple fact), and a structured version (giving data). For example, we can talk of a function f having the *property* of being computable (which is a simple fact), or we can talk about *computation structures* for f: programs computing f. Chapters 7 and 8 deal with these notions.

When we switch to a univalent approach, we are not abandoning the Curry-Howard interpretation, but augmenting it: we still have Σ and $+$, allowing us to express a logic of structures, but we also have a notion of proposition, for which we can introduce a logic of propositions, as we do in Section 2.2. This allows a richer expression of ideas, as we get to choose in our definitions and theorems whether to use structured or propositional notions. Often, one choice is more natural than the other.

As being a proposition is a property of types, we can introduce a type, and then prove after the fact that it is a proposition. There are two benefits of this: it gives us concise and clear language for expressing that structure is trivial, and more usefully, it gives us tools for turning simple facts into interesting structure. For an example of this latter situation, see Lemma 3.13, where we show an example where the truth of a proposition can allow us to compute a natural number.

The distinction between structure and property guides formalization: for example, we can ask for a *monoid structure* on a type:

A *monoid structure* on X consists of a multiplication operation $- \cdot - : X^2 \to X$ and a unit $e : X$, satisfying the equations

$$\begin{aligned} (\forall x, y, z \in X) &\quad x \cdot (y \cdot z) = (x \cdot y) \cdot z \\ (\forall x \in X) &\qquad\quad x \cdot e = e \cdot x = x \end{aligned}$$

Then we have the type family $\mathsf{MonStruct} : \mathcal{U} \to \mathcal{U}$. Using ideas from Chapter 2, we can also discuss the property of *being a monoid* $\mathsf{isMonoid} : \mathcal{U} \to \mathsf{Prop}$ using truncation, but this is of little use: there may be many possible monoid structures on a type M, but a witness of $\mathsf{isMonoid}(M)$ cannot distinguish between these: the property of being a monoid does not allow us to access an actual monoid structure, in general. On the other hand, given a monoid $(M, \cdot, e, -) : \mathsf{Mon}$—namely, a set equipped with a monoid structure—and an element $m : M$, we can define the type of inverses of an element

$$\mathsf{inverse}_M(m) : \Sigma(n : M), (m \cdot n = e) \times (n \cdot m = e),$$

and while $\mathsf{inverse} : M \to \mathcal{U}$ is a priori structure, we can in fact prove that $\mathsf{inverse}(m)$ is a proposition for all $m : M$. That is, we can show that the type of inverses is a proposition. In our case, this means that asserting the existence of an inverse is the same as giving an explicit inverse; or even more evocatively: inverse are uniquely specified. As we can also show that products over propositions are again propositions, we have a *predicate* $\mathsf{isGrp} : \mathsf{Mon} \to \mathsf{Prop}$, saying whether a monoid is a group. So while a group is structure imposed on a set, the structure is already imposed by the underlying monoid (in fact, already by the underlying semigroup).

The ability to infer that a type is a proposition distinguishes univalent mathematics from, for example, the calculus of (inductive) constructions [21, 25], which also has a universe of propositions. However, propositions are completely separate from other types in CiC, and no structural content can be inferred from them.

Unfortunately, pure MLTT does not give us everything we need to make use of univalent definitions properly. In particular, we cannot form a proposition-valued version of $\vee$ or $\exists$, and we cannot show that $\neg$ and $\forall$ preserve propositionhood. We will show how to extend MLTT with a truncation operator (resolving $\vee$ and $\exists$) and a principle of *function extensionality* (to resolve the problem with $\neg$ and $\forall$) as we develop the univalent approach in Chapter 2.

Properties—proposition-valued type families—play a central role in any work in a univalent framework. In Chapter 2 we will provide additional technical justification for the claim that propositions should be viewed as types with trivial structure, but we will make use of the fact in our exposition from here onward. We will see in the next section that we have the type-theoretic analog of the Leibniz law, that identical objects are indiscernible. Then if P is some proposition, all its elements will be indiscernible.

Consequently, once we know that a type P is a proposition, we will often leave elements of P unnamed. Moreover, we will ignore the behavior of constructions on propositions (see, for example, the definition of the lifting monad in Chapter 5). Finally, we will stick to a rigid naming convention: If we give a type family $A \to \mathcal{U}$ a name beginning with is, we will (at some point) prove that this type family is proposition-valued, and we will also have capital letters somewhere in the name—for example $\mathsf{isContr}$, isProp, isSet, $\mathsf{isGroupoid}$. Moreover, after proving $\mathsf{isXYZ} : A \to \mathcal{U}$ to be proposition-valued, when we need to name an element of $\mathsf{isXYZ}(a)$, we will often use the lowercase $\mathsf{isxyz} : \mathsf{isXYZ}(a)$.

1.7 Homotopies and equivalences

The development of univalent mathematics relies on tools for comparing functions and for comparing types which we explain here. The tools are those of *homotopy* between functions and *equivalence* between types. The logical equivalence defined earlier is sufficient when we are dealing with properties, but when comparing structures, it is insufficient: for example, the unit type is logically equivalent to the natural numbers. One of the first insights leading to univalent mathematics was the definition of *equivalence* in Voevodsky's Foundations library [91], which refines the traditional notion of isomorphism. Since then, several equivalent notions have been given, which are the focus of Chapter 4 of the HoTT Book. We are interested in three notions, which they call *quasi-invertibility* (Definition 2.4.6), *bi-invertibility* (Definition 4.3.1) and *contractibility* (Definition 4.4.1). The notion of quasi-invertibility is problematic; we will discuss why after formally introducing these notions. We diverge slightly from the HoTT Book in our terminology. In particular we call bi-invertibility *equivalence*, and a *quasi-inverse* of a function, simply an *inverse*. To express these notions, we need the notion of *homotopy*, or *pointwise equality*.

Definition 1.4. A *homotopy* between two functions $f, g : \Pi(x : X), B(x)$ is a pointwise equality between $f(x)$ and $g(x)$:

$$f \sim g \stackrel{\mathsf{def}}{=} \Pi(x : X), f(x) = g(x).$$

For any $f, g : \Pi(x : X), B(x)$ we have a map

$$\mathsf{happly} : (f = g) \to (f \sim g)$$

defined by path induction with $\mathsf{happly}(\mathsf{refl}) \stackrel{\mathsf{def}}{=} \lambda x.\, \mathsf{refl}$.

Definition 1.5. For $f : A \to B$, and $b : B$, the *fiber* of f over b is the type

$$\mathsf{fib}_f(b) \stackrel{\mathsf{def}}{=} \Sigma(a : A), f(a) = b.$$

Now we may define our notions of equivalence.

Definition 1.6. A function $f : A \to B$ is *contractible* when it *has contractible fibers*:

$$\mathsf{isContr}(f) \stackrel{\mathsf{def}}{=} \Pi(b : B), \mathsf{isContr}(\mathsf{fib}_f(b)).$$

A *right inverse* for f is a map $g : B \to A$ such that $f \circ g \sim \mathsf{id}_B$, and a *left inverse* for f is a map $g : B \to A$ such that $g \circ f \sim \mathsf{id}_A$. That is,

$$\mathsf{rinv}(f) \stackrel{\mathsf{def}}{=} \Sigma(g : B \to A), (f \circ g \sim \mathsf{id}_B).$$

$$\mathsf{linv}(f) \stackrel{\mathsf{def}}{=} \Sigma(g : B \to A), (g \circ f \sim \mathsf{id}_A)$$

The map f is an *equivalence* when it has a left inverse and a right inverse:

$$\mathsf{isEquiv}(f) \stackrel{\mathsf{def}}{=} \mathsf{linv}(f) \times \mathsf{rinv}(f).$$

We write $A \simeq B$ for the type of equivalence from A to B:

$$A \simeq B \stackrel{\mathsf{def}}{=} \Sigma(f : A \to B), \mathsf{isEquiv}(f).$$

When $f : A \to B$ has a left inverse, we say that A is a *retract* of B. We call a function which has a left inverse a *section* and a function which has a right inverse a *retraction*.

An *inverse* for f is a map $g : B \to A$ which is both a left and a right inverse for f:

$$\mathsf{inverse}(f) \stackrel{\mathsf{def}}{=} \Sigma(g : B \to A), ((f \circ g) \sim \mathsf{id}) \times ((g \circ f) \sim \mathsf{id}).$$

The distinction between equivalence and having an inverse is the first example of the difference between structure and property. We can show that $\mathsf{isEquiv}(f)$ is a proposition—that is, that being an equivalence is property—but this is not true in general for the type $\mathsf{inverse}(f)$—that is, having an inverse is structure. Lemma 4.1.1 of the HoTT Book shows that if $\mathsf{inverse}(f)$ (which they call $\mathsf{qinv}(f)$) has an element, where $f : A \to B$, then

$$\mathsf{inverse}(f) \simeq \Pi(x : A), x = x.$$

The fact that having an inverse is not a proposition means we cannot naively use $\mathsf{inverse}(f)$ as a replacement for $\mathsf{isEquiv}(f)$. In particular, we will shortly define a map $\mathsf{idtoequiv} : A = B \to A \simeq B$, taking a path in the universe to an equivalence, and the univalence axiom says that $\mathsf{idtoequiv}$ is an equivalence, but the corresponding statement, that the map $(X = Y) \to \Sigma(f : X \to Y), \mathsf{inverse}(f)$ has an inverse is actually inconsistent, since a consequence of this statement is that $\mathsf{inverse}(f)$ is a proposition. We will see this in detail when we talk about the univalence axiom on Section 2.4

We will show in Section 1.9 that having an inverse, being contractible and being an equivalence are all logically equivalent after we develop some tools for working with paths. In the meantime, we fulfill our promise to justify the phrase "logical equivalence". We aim to use propositions to encode logic in our system, so the information encoded directly by propositions is logical information. This means that when we call two propositions logically equivalent, we should be able to conclude that they are equivalent. Indeed, we have the following.

Lemma 1.7. *If A and B are propositions, then $(A \Leftrightarrow B) \to (A \simeq B)$.*

Proof. Suppose $f : A \to B$ and $g : B \to A$. We have that $g(f(a)) = a$ and $f(g(b)) = b$, as A and B are propositions. $\square$

The words *path*, *homotopy*, *contractible* and *equivalence* come from homotopy theory. The modern terminology for several other important notions come also from homotopy theory. Nevertheless, no knowledge of homotopy theory is needed to understand our type theory. The next name coming from homotopy is *application on paths*, which expresses that functions respect equality. Given any function $f : A \to B$ and $x, y : A$, there is a function

$$\mathsf{ap}_f : x = y \to f(x) = f(y)$$

defined by path induction with

$$\mathsf{ap}_f(\mathsf{refl}_x) \stackrel{\mathsf{def}}{=} \mathsf{refl}_{f(x)}.$$

In involved computations, we may abuse notation and write $f(p)$ instead of $\mathsf{ap}_f(p)$. Application on paths tells us that functions are *congruences*, or act *functorially* with respect to equality. Formally, we mean the following.

Lemma 1.8. *For any $f : A \to B$ we have*

1. $\mathsf{ap}_f(\mathsf{refl}_x) = \mathsf{refl}_{f(x)}$;
2. $\mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$;
3. $\mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$.

where $p : x = y$ and $q : y = z$ for some $x, y, z : A$.

Proof. The first equation is definitional, the second and third follow by path induction: First, fix $p \equiv q \equiv \mathsf{refl}_x$, so that

$$\mathsf{ap}_f(\mathsf{refl} \cdot \mathsf{refl}) = \mathsf{ap}_f(\mathsf{refl}) = \mathsf{refl} = \mathsf{refl} \cdot \mathsf{refl} = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q).$$

Then, fix $p \equiv \mathsf{refl}_x$ so that we have

$$\mathsf{ap}_f(\mathsf{refl}^{-1}) = \mathsf{ap}_f(\mathsf{refl}) = \mathsf{refl} = \mathsf{refl}^{-1} = \mathsf{ap}_f(\mathsf{refl})^{-1}. \qquad \square$$

Note that ap immediately gives us that any retract of a proposition or contractible type is again a proposition (or contractible type).

Theorem 1.9. *If $f : A \to B$ and $(g, \eta) : \mathsf{linv}(f)$ is a left inverse of f, then $\mathsf{isContr}(B) \to \mathsf{isContr}(A)$ and $\mathsf{isProp}(B) \to \mathsf{isProp}(A)$.*

Proof. Letting $c : B$ be the center of contraction of B with $w : \Pi(b : B), c = b$, we have for any $a : A$,

$$g(c) \overset{\mathsf{ap}_g(w(f(a)))}{=} gf(a) \overset{\eta(a)}{=} a,$$

so $g(c)$ is the center of contraction of A.

Similarly, if $w : \mathsf{isProp}(B)$ and $a, b : A$, then

$$a \overset{\eta(a)^{-1}}{=} gf(a) \overset{\mathsf{ap}_{gf}(w(a,b))}{=} gf(b) \overset{\eta(b)}{=} b,$$

so A is a proposition. □

A direct corollary is that propositions (and singletons) are closed under equivalence.

Corollary 1.10. *If A is a retract of B and B is a proposition, then A and B are equivalent.*

Proof. As A is a retract of B, we have $A \Leftrightarrow B$ and A is a proposition. Then $A \simeq B$ by Lemma 1.7. □

This allows us to give the following characterization of the contractible types as propositions with elements. 1.

Lemma 1.11. *The following are logically equivalent for any type A:*

- *$A \simeq 1$; (A is equivalent to 1)*
- *$\mathsf{isContr}(A)$; (A is contractible)*
- *$A \times \mathsf{isProp}(A)$; ($A$ is a proposition with an inhabitant)*

Proof. If $f : A \simeq 1$ then we know $c \overset{\mathsf{def}}{=} f^{-1}(\star) : A$, where $f^{-1} : 1 \to A$ is an inverse of f. For $a : A$, we have $f(a) = \star$, so $a = f^{-1}(f(a)) = f^{-1}(\star) = c$. So then A is contractible.

Let A be contractible, with center of contraction $c : A$. For $a, b : A$, we have $a = c = b$, so A is a proposition.

If $a : A$, we have a map $c_a : 1 \to A$ given by $c_a(\star) \overset{\mathsf{def}}{=} a$, and we have $c_\star : A \to 1$. If A is additionally a proposition, then we have $A \Leftrightarrow 1$, and so $A \simeq 1$. □

Corollary 1.12. *All contractible types are equivalent.*

For dependent function $f : \Pi(x : A), B(x)$ the situation is more subtle. As $f(x)$ and $f(y)$ need not even have the same type, the expression $f(x) = f(y)$ is not even well-formed. To resolve this issue, we notice that paths give rise to functions, called *transport functions*. For each $x, y : A$, we define by path induction

$$\begin{aligned}&\mathsf{transport}^B : (x = y) \to B(x) \to B(y)\\&\mathsf{transport}^B(\mathsf{refl}, u) \stackrel{\mathsf{def}}{=} u,\end{aligned}$$

and then given $f : \Pi(x : A), B(x)$ we have again by path induction, for each $x, y : A$, a *dependent application* function,

$$\begin{aligned}&\mathsf{apd}_f : \Pi(p : x = y), \mathsf{transport}(p, f(x)) = f(y)\\&\mathsf{apd}_f(\mathsf{refl}_x) \stackrel{\mathsf{def}}{=} \mathsf{refl}_{f(x)}.\end{aligned}$$

We will call a path $q : \mathsf{transport}^B(p, u) = v$ a path *in B lying over p*, and write

$$(u =^p_B v) \stackrel{\mathsf{def}}{=} \mathsf{transport}^B(p, u) = v,$$

dropping the subscript when the type family is clear from context. Then for $p : x = y$ we may write

$$\mathsf{apd}_f(p) : f(x) =^p f(y).$$

The spatial picture giving rise to the homotopy-theoretic names is as follows: A type family $B : A \to \mathcal{U}$ can be seen as a space lying over A, with $B(a)$ the fiber over A. A dependent function gives a section of B, and using this, we can lift any path $p : x \rightsquigarrow y$ in A to a path in the total space of B. This path can be decomposed using a natural family of paths $B(x) \rightsquigarrow B(y)$ and a path in $B(y)$. This natural family of paths is captured by $\mathsf{transport}$, while the path in $B(y)$ is apd.

There is a similar logical picture, which is captured more directly by the above definitions: Via ap_B, an equality $p : x = y$ in A gives rise to an equality $B(x) = B(y)$ in the universe. As an equality between types allows us to identify the types, there should be a function $B(x) \to B(y)$, which we call the function *induced by* p. Indeed, we have for any $A, B : \mathcal{U}$ the map

$$\mathsf{idtofun} : (A = B) \to (A \to B)$$

defined by path induction with

$$\mathsf{idtofun}(\mathsf{refl}_A) \stackrel{\mathsf{def}}{=} \mathsf{id}_A\,.$$

Then, applying this to $\mathsf{ap}_B : (x = y) \to (B(x) = B(y))$, and we have

$$\mathsf{transport}(p, u) = \mathsf{idtofun}(\mathsf{ap}_B(p), u).$$

Conversely, we may define $\mathsf{idtofun}$ from $\mathsf{transport}$ as

$$\mathsf{idtofun}(p, u) \stackrel{\mathsf{def}}{=} \mathsf{transport}^{\mathsf{id}_{\mathcal{U}}}(p, u).$$

In any case, by path induction, we can show that the map $\mathsf{idtofun}(p)$ is an equivalence for any $p : A = B$: for $\mathsf{refl}_A : A = A$, we have that the identity function $\mathsf{id}_A : A \to A$ is an equivalence, as it is its own inverse:

$$((\mathsf{id}_A, \lambda x.\mathsf{refl}), (\mathsf{id}_A, \lambda x.\mathsf{refl})) : \mathsf{isEquiv}(\mathsf{idtofun}(\mathsf{refl})).$$

From this and path induction we get a map $\mathsf{idtoequiv} : (A = B) \to (A \simeq B)$. In other words equal types are equivalent. Moreover, viewing type families as encoding properties, the type of transport expresses Leibniz's law—the indescernibility of identicals. The function apd tells us that Leibniz's law respects choice of witness.

We now have the definitions we need to show that $\mathbb{N}$ is a set.

Theorem 1.13. *The type* $\mathbb{N}$ *is a set, and is not a proposition.*

We use a modification of a technique called *encode-decode* which is typically used to characterize the type of paths in a space. Encode-decode proofs proceed by defining a *coding family*

$$\mathsf{code} : \mathbb{N} \to \mathbb{N} \to \mathcal{U},$$

together with a family of *encoding functions*,

$$\mathsf{encode} : \Pi(x, y : \mathbb{N}), (x = y) \to \mathsf{code}(x, y),$$

and *decoding functions*

$$\mathsf{decode} : \Pi(x, y : \mathbb{N}), \mathsf{code}(x, y) \to (x = y),$$

such that $\mathsf{encode}_{x,y}$ and $\mathsf{decode}_{x,y}$ are inverses.

In our case, it suffices to show that $x = y$ is a retract of $\mathsf{code}(x, y)$ and that code is a predicate.

Proof. Define the coding family

$$\begin{aligned} \mathsf{code}(0, 0) &\stackrel{\mathsf{def}}{=} 1; \\ \mathsf{code}(0, x + 1) &\stackrel{\mathsf{def}}{=} \emptyset; \\ \mathsf{code}(x + 1, 0) &\stackrel{\mathsf{def}}{=} \emptyset; \\ \mathsf{code}(x + 1, y + 1) &\stackrel{\mathsf{def}}{=} \mathsf{code}(x, y). \end{aligned}$$

In order to define the encoding functions we define a function $r : \Pi(x : \mathbb{N}), \mathsf{code}(x, x)$ which gives the encoding of the reflexivity path,

$$\begin{aligned} r_0 &\stackrel{\mathsf{def}}{=} \star; \\ r_{n+1} &\stackrel{\mathsf{def}}{=} r_n. \end{aligned}$$

Then we have encoding functions

$$\mathsf{encode}_{m,n}(p) \stackrel{\mathsf{def}}{=} \mathsf{transport}^{\mathsf{code}(m,-)}(p, r_m),$$

and decoding functions

$$\begin{aligned}
\mathsf{decode}_{0,0} &\stackrel{\mathsf{def}}{=} \lambda w.\mathsf{refl}_0;\\
\mathsf{decode}_{0,x+1} &\stackrel{\mathsf{def}}{=} ! : \emptyset \to (0 = x+1);\\
\mathsf{decode}_{x+1,0} &\stackrel{\mathsf{def}}{=} ! : \emptyset \to (x+1 = 0);\\
\mathsf{decode}_{x+1,y+1} &\stackrel{\mathsf{def}}{=} \mathsf{ap}_{\mathsf{succ}} \circ \mathsf{decode}_{x,y}.
\end{aligned}$$

To show that $\mathsf{decode} \circ \mathsf{encode}$ is the identity, we may use path induction. We have

$$\mathsf{decode}_{x,x}(\mathsf{encode}_{x,x}(\mathsf{refl})) = \mathsf{decode}(\mathsf{transport}^{\mathsf{code}(x,-)}(\mathsf{refl}, r_x)) = \mathsf{decode}_{x,x}(r_x).$$

By induction on x, we see that $\mathsf{decode}(r_x) = \mathsf{refl}_x$: this is by definition at 0. We have that $\mathsf{ap}_{\mathsf{succ}}(\mathsf{refl}_x) = \mathsf{refl}_{x+1}$ by definition of ap, so then for the successor case we have

$$\mathsf{decode}_{x+1,x+1}(r_{x+1}) = \mathsf{ap}_{\mathsf{succ}}(\mathsf{decode}_{x,x}(r_{x+1})) = \mathsf{ap}_{\mathsf{succ}}(\mathsf{decode}(r_x)) = \mathsf{ap}_{\mathsf{succ}}(\mathsf{refl}_x) = \mathsf{refl}_{x+1}.$$

To show that $\mathsf{code}(x, y)$ is a proposition, we proceed by induction on x and y. We have that $\mathsf{code}(0, 0) \equiv 1$ and $\mathsf{code}(x, y+1) = \mathsf{code}(x+1, y) = \emptyset$. Finally we have

$$\mathsf{code}(x+1, y+1) \equiv \mathsf{code}(x, y),$$

and $\mathsf{code}(x, y)$ is a proposition by the inductive hypothesis.

We have that $x = y$ is a retract of a proposition, so by Theorem 1.9, $x = y$ is a proposition for any $x, y : \mathbb{N}$. Since we also have

$$\lambda w.w(0,1) : \big(\Pi(x, y : \mathbb{N}), x = y\big) \to (0 = 1),$$

and that $0 \neq 1$, we have $\neg\,\mathsf{isProp}(\mathbb{N})$. $\square$

In the above proof we got lucky: we defined encode using $\mathsf{transport}$, but in the end we didn't need to compute the behavior of transport on anything besides the reflexivity paths. This will not always suffice, so in the next section we discuss some key results concerning transport and identity types.

1.8 Transport, identities and the basic types

Identities, and transport along them, are of core importance, so we need some basic results telling us how to compute with them. The proofs of most of these results are direct application of path induction which we do not try to motivate here. For our purposes, the results in this section are technical lemmas characterizing the behavior of identity types and $\mathsf{transport}$. More detailed discussion (and in some cases, more detailed proofs) can be found in Chapter 2 of the HoTT Book.

We start with an important characterization of equality in dependent sum types.

Theorem 1.14. *Let $B : A \to \mathcal{U}$ be a type family, and let $w, w' : \Sigma(a : A), B(a)$. Then there is an equivalence*

$$(w = w') \simeq \big(\Sigma(p : \mathsf{pr}_0(w) = \mathsf{pr}_0(w')), \mathsf{transport}(p, \mathsf{pr}_1(w)) = \mathsf{pr}_1(w')\big).$$

Proof. Let $S \overset{\mathsf{def}}{=} \Sigma(x : A), B(x)$ and let

$$\mathsf{code}_{w,w'} \overset{\mathsf{def}}{=} \Sigma(p : \mathsf{pr}_0(w) = \mathsf{pr}_0(w')), \mathsf{transport}(p, \mathsf{pr}_1(w)) = \mathsf{pr}_1(w')$$

and define the family of maps

$$\mathsf{encode} : \Pi(w, w' : S), (w = w') \to \mathsf{code}_{w,w'}$$

by path induction with

$$\mathsf{encode}_{w,w}(\mathsf{refl}_w) \overset{\mathsf{def}}{=} (\mathsf{refl}_{\mathsf{pr}_0(w)}, \mathsf{refl}_{\mathsf{pr}_1(w)}).$$

Now consider $w, w' : S$ and $r : \mathsf{code}_{w,w'}$. By induction on all three pairs w, w' and r, we may assume $w = (a, b)$ and $w' = (a', b')$ and $r = (p, q)$ with $p : a = a'$ and $q : b =^p_B b'$. We can perform path induction on p, with $a \equiv a'$ and $p \equiv \mathsf{refl}$, so that $q : b =_{B(a)} b'$. Again, by path induction, we may treat q as refl_b. Then we know that $\mathsf{refl}_{(a,b)} : w = w'$. So we have a family of functions

$$\mathsf{decode} : \Pi(w, w' : S), \mathsf{code}_{w,w'} \to (w = w').$$

such that

$$\mathsf{decode}((a, b), (a, b), (\mathsf{refl}_a, \mathsf{refl}_b)) \equiv \mathsf{refl}_{(a,b)}.$$

By induction, proving encode and decode to be inverse reduces to proving that

$$\mathsf{decode}(\mathsf{encode}(\mathsf{refl}_{(a,b)})) = \mathsf{refl}_{(a,b)}$$

and that

$$\mathsf{encode}(\mathsf{decode}(\mathsf{refl}_a, \mathsf{refl}_b)) = (\mathsf{refl}_a, \mathsf{refl}_b).$$

Both of these equations hold definitionally. □

We will abuse notation write (p, q) for the path $(a, b) = (a', b')$ arising from $\mathsf{decode}(p, q)$.

Corollary 1.15. *For any $w : \Sigma(a : A), B(a)$, we have $w = (\mathsf{pr}_0(w), \mathsf{pr}_1(w))$.*

Theorem 1.16. *Let $B : A \to \mathcal{U}$ and let $C : (\Sigma(a : A), B(a)) \to \mathcal{U}$. Define $C' : A \to \mathcal{U}$ by*

$$C'(a) \stackrel{\mathsf{def}}{=} \Sigma(b : B(a)), C(a, b).$$

For any $p : a = a'$ and any pair $(b, c) : \Sigma(b : B(a)), C(a, b)$, we have

$$\mathsf{transport}^{C'}(p, (b, c)) = (\mathsf{transport}^{B}(p, b), \mathsf{transport}^{C}((p, \mathsf{refl}), c)).$$

In other words, transport works pairwise.

Proof. By path induction, it is enough to show

$$\mathsf{transport}^{C'}(\mathsf{refl}, (b, c)) = (\mathsf{transport}^{B}(\mathsf{refl}, b), \mathsf{transport}^{C}((\mathsf{refl}, \mathsf{refl}), c)).$$

As we know that $(\mathsf{refl}, \mathsf{refl}) \equiv \mathsf{refl}$, the definition of transport tells us that this is asking for

$$(b, c) = (b, c),$$

which we have by reflexivity. $\square$

Lemma 1.17. *Given a type $Y : \mathcal{U}$, a path $p : X = X'$ between types X and X', and a function $f : X \to Y$, we have*

$$\mathsf{transport}^{\lambda X.X \to Y}(p, f) = \mathsf{idtofun}(p) \circ f.$$

Proof. By path induction, we need to see that

$$\mathsf{transport}^{\lambda X.X \to Y}(\mathsf{refl}, f) = \mathsf{idtofun}(\mathsf{refl}) \circ f.$$

Applying the definition of transport and idtofun, it suffices to see that $f = \mathsf{id} \circ f$, which follows from the η rule. $\square$

Theorem 1.18. *Given a function $f : A \to B$, a family $C : B \to \mathcal{U}$, a path $p : x =_A y$ and $c : C(f(x))$ we have*

$$\mathsf{transport}^{C \circ f}(p, c) = \mathsf{transport}^{C}(\mathsf{ap}_f(p), c).$$

Proof. By path induction, we need to see that

$$\mathsf{transport}^{C \circ f}(\mathsf{refl}, c) = \mathsf{transport}^{C}(\mathsf{ap}_f(\mathsf{refl}), c).$$

This reduces to

$$c = c,$$

which we have by reflexivity. $\square$

Theorem 1.19. *For any $A : \mathcal{U}$ and $a : A$ and $p : x =_A y$, we have*

$$\begin{aligned}
\mathsf{transport}^{\lambda x.a=x}(p, q) &= q \cdot p && \text{when } q : a = x, \\
\mathsf{transport}^{\lambda x.x=a}(p, q) &= p^{-1} \cdot q && \text{when } q : x = a, \\
\mathsf{transport}^{\lambda x.x=x}(p, q) &= p^{-1} \cdot q \cdot p && \text{when } q : x = x.
\end{aligned}$$

This is a special case of the following theorem characterizing transport along identity types in a type family, by taking $B = \lambda x.A$ and f, g to be constant functions.

Theorem 1.20. *Let $f, g : \Pi(a : A), B(a)$ with $p : a = a'$ and $q : f(a) = g(a)$. Then,*

$$\mathsf{transport}^{\lambda x.f(x)=g(x)}(p, q) = (\mathsf{apd}_f(p))^{-1} \cdot \mathsf{ap}_{\mathsf{transport}^B(p)}(q) \cdot \mathsf{apd}_g(p).$$

Proof. By path induction on p, we need to see

$$\mathsf{transport}^{\lambda x.f(x)=g(x)}(\mathsf{refl}, q) = (\mathsf{apd}_f(\mathsf{refl}))^{-1} \cdot \mathsf{ap}_{\mathsf{transport}^B(\mathsf{refl})}(q) \cdot \mathsf{apd}_g(\mathsf{refl}).$$

This reduces to

$$q = \mathsf{ap}_{\mathsf{id}}(q),$$

and it is immediate that $\mathsf{ap}_{\mathsf{id}}$ is homotopic to the identity. □

1.9 Equivalences and contractible fibers

To conclude this chapter, we show that having contractible fibers, having an inverse, and being an equivalence are logically equivalent notions. This section therefore provides an important example of how to use the machinery in the previous section.

Theorem 1.21. *For any $f : A \to B$, the following types are logically equivalent:*

1. $\mathsf{inverse}(f)$,
2. $\mathsf{isEquiv}(f)$,
3. $\Pi(b : B), \mathsf{isContr}(\mathsf{fib}_f(b))$.

To relate contractibility to equivalence, we need to spend some time examining fibers. As the fiber is a sum over a family valued in path types, we first give two important lemmas about families of paths.

Lemma 1.22. *For any type A and any $a : A$, the type $\Sigma(x : A), x = a$ is contractible.*

This type $\Sigma(x : A), x = a$ is sometimes called the *singleton (based) at a*.

Proof. We wish to have (a, refl_a) as the center of contraction, so we need to see that for any $x : A$ and $p : x = a$ we have

$$(x, p) = (a, \mathsf{refl})$$

which we can show by based path induction. We need only see that $(a, \mathsf{refl}) = (a, \mathsf{refl})$, which we have by $\mathsf{refl}_{(a,\mathsf{refl})}$. □

We could also prove this by a direct computation using the machinery from the previous section. For any $(x, p) : \Sigma(x : A), x = a$ we have

$$\big((x, p) = (a, \mathsf{refl})\big) \simeq \Big(\Sigma(q : x = a), \mathsf{transport}^{-=a}(p, q) = \mathsf{refl}\Big).$$

By Theorem 1.19, we have that $\mathsf{transport}^{-=a}(p, p) = p^{-1} \cdot p = \mathsf{refl}$.

Lemma 1.22 can be read as the (classically trivial) claim that $\{a\}$ is equivalent to the type $\{x \in A \mid x = a\}$. It will be used frequently and is relevant here because it tells us that the identity function has contractible fibers:

Lemma 1.23. *For any $A : \mathcal{U}$, the identity function $\mathsf{id}_A : A \to A$ has contractible fibers.*

Proof. The fiber of id at $a : A$ is the singleton based at a. □

In fact, the equivalence in Lemma 1.22 lifts to any type family over A.

Lemma 1.24. *For any type family $B : A \to \mathcal{U}$ and any $a : A$,*

$$B(a) \simeq \Sigma((a', b') : \Sigma(a' : A), B(a')), a' = a.$$

Proof. The right-hand side is equivalent, by a reshuffling map, to

$$B \overset{\mathsf{def}}{=} \Sigma(a' : A), B(a') \times (a = a').$$

There is a map $B \to B(a)$ given by $(a', b', p) \mapsto \mathsf{transport}^B(p, b')$, with a candidate inverse $b \mapsto (a, b, \mathsf{refl})$. Since $\mathsf{transport}(\mathsf{refl}, b) = b$, the type $B(a)$ is a retract of B. Now, we need to see for any $(a', b', p) : B$ that

$$(a', b', p) = (a, \mathsf{transport}(p, b'), \mathsf{refl}).$$

We have by assumption $p : a' = a$, and we know $\mathsf{refl} : \mathsf{transport}(p, b') = \mathsf{transport}(p, b')$. Finally, we need to see that $\mathsf{transport}^{-=a'}(p, p) = \mathsf{refl}$, which is an application of Theorem 1.19. $\square$

We can generalize this one step further.

Lemma 1.25. *For any $a : A$ and $B : \Pi(x : A), x = a \to \mathcal{U}$ the map*

$$e : B(a, \mathsf{refl}) \to \Sigma(x : A), \Sigma(p : x = a), B(x, p)$$

given by

$$b \mapsto (a, \mathsf{refl}, b)$$

is an equivalence.

Proof. The inverse is given by

$$f(x, p, b) \overset{\mathsf{def}}{=} \mathsf{transport}((\mathsf{refl}, w), b)$$

where $w : \mathsf{transport}^{-=a}(p, p) = \mathsf{refl}$ comes from Theorem 1.19. $\square$

The logical equivalence between having an inverse and being an equivalence is direct, and by looking at the center of contraction of $\mathsf{fib}_f(b)$, when f is contractible, we can construct an inverse for f. The difficulty in Theorem 1.21 is in showing that a map with an inverse has contractible fibers. The key insight is to relate the fibers of a section s to the fibers of the map $B \to B$ given by composing with its retraction.

Lemma 1.26. *Given $r : B \to A$ and $s : A \to B$ together with a homotopy $\eta : r \circ s \sim \mathsf{id}_A$, for any $b : B$ we have that $\mathsf{fib}_s(b)$ is a retract of $\mathsf{fib}_{s \circ r}(b)$.*

Proof. The map

$$\overline{s} : \big(\Sigma(a : A), s(a) = b\big) \to \big(\Sigma(b' : B), sr(b') = b\big)$$

is given by

$$\overline{s}(a, p) \overset{\mathsf{def}}{=} (s(a), \mathsf{ap}_s(\eta_a) \cdot p),$$

The second component is the path

$$s(r(s(a))) \overset{\mathsf{ap}_s(\eta_a)}{=} s(a) \overset{p}{=} b.$$

The candidate retraction

$$\overline{r} : \big(\Sigma(b' : B), sr(b') = b\big) \to \big(\Sigma(a : A), s(a) = b\big)$$

is given by

$$\overline{r}(b', q) \overset{\mathsf{def}}{=} (r(b'), q).$$

We need to see that $(a, p) = \overline{r}(\overline{s}(a, p))$. By Theorem 1.14 we need to see that there is some $w : a = r(s(a))$ such that $\mathsf{transport}^{\lambda a.s(a)=b}(w, p) = \mathsf{ap}_s(\eta_a) \cdot p$. We have $\eta_a^{-1} : a = r(s(a))$, so we compute,

$$\begin{aligned}
\mathsf{transport}^{\lambda a.s(a)=b}(\eta_a^{-1}, p) &= \mathsf{transport}^{\lambda b'.b'=b}(\mathsf{ap}_s(\eta_a^{-1}), p) && \text{by Theorem 1.18} \\
&= (\mathsf{ap}_s(\eta_a^{-1}))^{-1} \cdot p && \text{by Theorem 1.19} \\
&= (\mathsf{ap}_s(\eta_a)) \cdot p && \text{by Lemma 1.8.} \quad \square
\end{aligned}$$

Finally, we may prove Theorem 1.21.

Proof (of Theorem 1.21). Given $(g, (\eta, \epsilon))$: $\mathsf{inverse}(f)$ we have $((g, \eta), (g, \epsilon))$: $\mathsf{isEquiv}(f)$. Conversely, if $\eta : gf \sim \mathsf{id}_A$ and $\epsilon : fh \sim \mathsf{id}_B$, then for any $b : B$ we have $g(b) = g(f(h(b))) = h(b)$, and this gives us a homotopy $\theta : fg \sim \mathsf{id}_B$. Explicitly, θ_b is given by the composite path

$$f(g(b)) \overset{\mathsf{ap}_{f \circ g}(\epsilon_b^{-1})}{=} f(g(f(h(b)))) \overset{\mathsf{ap}_f(\eta_{h(b)})}{=} f(h(b)) \overset{\eta_b}{=} b,$$

and so $(g, (\eta, \theta))$ gives an inverse of f.

The implication from 3. to 1. is only slightly more involved: Define $g : B \to A$ to be the function giving the first component of the center of contraction of $\mathsf{fib}_f(b)$. In other words, for any $b : B$ we have by assumption an element

$$((a, p), w) : \mathsf{isContr}(\mathsf{fib}_f(b)),$$

and we may define $g(b) \overset{\mathsf{def}}{=} a$. Note that $p : f(g(b)) = b$. Moreover, we know that we have (a, refl) : $\mathsf{fib}_f\, f(a)$, and so we have $a = g(f(a))$ by Theorem 1.14 and the fact that $\mathsf{fib}_f(a)$ is contractible. Then we have that g is an inverse of f.

The implication from 1. to 3. is less obvious. As g is a section of f, we have that $\mathsf{fib}_g(b)$ is a retract of $\mathsf{fib}_{gf}(b)$, by Lemma 1.26. Since a retract of a contractible type is again contractible, it suffices to show that $\mathsf{fib}_{gf}(b)$ is contractible. However, since we have $g \circ f \sim \mathsf{id}_B$, we have an equivalence

$$\mathsf{fib}_{g \circ f}(b) \simeq \mathsf{fib}_{\mathsf{id}_B}(b),$$

and the latter is contractible by Lemma 1.23. □

We will see later that being an equivalence and having contractible fibers are in fact equivalent as types.

Before we finish the chapter, we state two results about families of maps; these are Theorems 4.7.6 and 4.7.7 of the HoTT Book. Let $P, Q : A \to \mathcal{U}$ and $f : \Pi(x : A), P(x) \to Q(x)$. Define the *total map* of f by

$$f_\Sigma : (\Sigma(x : A), P(x)) \to (\Sigma(x : A), Q(x))$$
$$f_\Sigma(x, p) \stackrel{\mathsf{def}}{=} (x, f(x, p)).$$

Theorem 1.27. *For any $f : \Pi(x : A), P(x) \to Q(x)$, any $a : A$ and $q : Q(a)$, we have*

$$\mathsf{fib}_{f(a)}(q) \simeq \mathsf{fib}_{f_\Sigma}(a, q).$$

Proof. Expanding the definition of fib_{f_Σ} and reshuffling gives an equivalence

$$\mathsf{fib}_{f_\sigma}(a, q) \simeq \Sigma(x : X), \Sigma(u : x = a), \Sigma(p : P(x)), \mathsf{transport}(u, f(x, p)) = q.$$

By Lemma 1.25, this is equivalent to

$$\Sigma(p : P(a)), \mathsf{transport}(\mathsf{refl}, f(a, p)) = q.$$

In turn this is equivalent to

$$\Sigma(p : P(a)), f(a, p) = q,$$

which is $\mathsf{fib}_{f(a)}(q)$ by definition. □

Theorem 1.28. *For any $f : \Pi(x : A), P(x) \to Q(x)$ we have that each $f(a)$ is an equivalence iff the map f_Σ is an equivalence.*

Proof. Suppose that $f(a)$ is an equivalence for each $a : A$. Let $(x, q) : \Sigma(x : A), Q(x)$. By the previous theorem, $\mathsf{fib}_{f_\Sigma}(x, q) \simeq \mathsf{fib}_{f(x)}(q)$, and the latter is contractible since $f(x)$ is an equivalence.

Conversely, let f_Σ be an equivalence and fix $a : X$ and let $q : Q(x)$. Then $\mathsf{fib}_{f(a)}(q)$ is equivalent to $\mathsf{fib}_{f_\sigma}(a, q)$, which is contractible since f_Σ is an equivalence. □

1.10 Discussion

A key point in the univalent perspective is the careful distinction between structure and property, while taking the former to subsume the latter. This distinction is present in logical systems (such as CZF or the Mitchell-Benabou language), but properties (formulas) are completely distinct from structures, which are captured by certain objects of the system. The idea of treating property as being subsumed by structure is not obvious from this perspective—indeed it is not clear that there is a way to give a purely logical system capturing the univalent perspective. Even in type-theoretic contexts, where there has always been some idea that propositions are types [57], the idea that being a proposition can be an internal statement (a predicate or structure), rather than a judgement is relatively recent. This view arose in type theory in conjunction with the consideration of extensionality; *bracket types* [65], which are used in the first approach to this way of distinguishing propositions [4], arose as a way to handle proof-irrelevance and intensionality.

Univalent mathematics is not the only, or the first, type theoretic approach to isolate a type of propositions. The calculus of inductive constructions [21, 21, 41] (on which Coq is based) is the canonical example. However, in CiC being a proposition is a judgement, meaning there is no internal statement correspond to $\mathsf{isProp}(P)$. Moreover, there is no computational content contained in the propositions of CiC. In short, propositions fit somewhat more naturally in the univalent world than in other type-theoretic foundational systems.

CHAPTER 2

Univalent mathematics

We now develop the univalent perspective. Since we use identity types rather than explicit equivalence relations (as we did in Section 1.2), we need extensionality principles—ways to prove that types are equal. Pure MLTT gives us no methods for this, so we must extend our theory. We begin with an examination of *function extensionality*, which allows us to conclude that two functions are equal when they are pointwise equal (Section 2.1). We then switch gears and examine propositional logic in our system (Section 2.2). In order to properly develop logic, we need a *truncation* operator, taking a type to its best representation as a proposition. This is analogous to the truncation operator we mentioned for toposes, but in this case we actually use it as a part of our language.

Before moving on to types of higher homotopy level (Section 2.6), we examine an extensionality principle for propositions and the *univalence axiom* which generalizes this to arbitrary types (Section 2.4); in order to properly discuss univalence, we first need to take a closer look at functions (Section 2.3). We end the chapter with some discussion of how to use *higher-inductive types*–types generated not only by constructors giving elements of the type, but also by *path constructors* giving paths between them (Section 2.7). This final section and the earlier Section 2.5 discuss principles which can be used, among other things, to define the truncation operator we assume in Section 2.2.

There is a subtlety with the extensionality principles we consider in this chapter: all of them depend on a universe $\mathcal{U}$. Until we explicitly introduce them as assumptions, we will treat them as properties of a universe.

2.1 Function extensionality

One of the difficulties of working in MLTT is that MLTT gives almost no methods for proving equalities. This first becomes apparent when we wish to show that two functions $f, g : A \to B$ are equal. The only equalities between functions arise as reflexivity out of judgmental equalities. For example consider the function $f : \mathbb{N} \to \mathbb{N}$ defined by recursion with

$$
\begin{aligned}
f(0) &= 0 \\
f(\mathsf{succ}(n)) &= \mathsf{succ}(f(n)).
\end{aligned}
$$

It is not possible to prove that $f = \mathsf{id}_{\mathbb{N}}$, even though we can prove that $\Pi(n : \mathbb{N}), f(n) = n$, and moreover we can show that for any constructor of $\mathbb{N}$ that this equality holds judgmentally. We expect f to equal $\mathsf{id}_{\mathbb{N}}$—we expect two functions to be equal when they are pointwise equal. In this section we will posit an axiom of *function extensionality* which tells us that this is the case.

An *axiom* in type theory is an element of a particular type that is declared to exist by fiat, independent of the relevant introduction and elimination rules. That axioms do not interact with introduction and elimination rules makes them potentially dangerous. For example, we could extend MLTT with an axiom $a : \mathbb{N}$. Then, after defining the predecessor function $\mathsf{pred} : \mathbb{N} \to \mathbb{N}$, we would have that $\mathsf{pred}(a) : \mathbb{N}$, but we cannot reduce this or compute with it in any way.

However, as we only work up to propositional equality, if we can prove that the type A is a proposition, then positing an element $a : A$ leads to no problems: For any $b : A$ that we can construct, we have $a = b$, and so we can use $\mathsf{transport}$ to interchange between a and b. So when we postulate function extensionality, we wish to ensure that the type representing the axiom is a proposition. We consider four logically equivalent formulations of function extensionality below. All but one can be shown to be a proposition.

Theorem 2.1. *For any universe $\mathcal{U}$, the following are logically equivalent:*

F1 *For any $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ and any $f, g : \Pi(a : A), B(a)$, the map $\mathsf{happly}_{f,g} : f = g \to f \sim g$ is an equivalence.*

F2 *For any $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ and any $f, g : \Pi(a : A), B(a)$, there is a map $f \sim g \to f = g$.*

F3 *The product of any family of propositions is a proposition: for any* $A : \mathcal{U}$ *and* $P : A \to \mathcal{U}$*,*

$$\big(\Pi(a : A), \mathsf{isProp}(P(a))\big) \to \mathsf{isProp}(\Pi(a : A), P(a)).$$

F4 *The product of any family of contractible types is contractible: for any* $A : \mathcal{U}$ *and* $P : A \to \mathcal{U}$*,*

$$\big(\Pi(a : A), \mathsf{isContr}(P(a))\big) \to \mathsf{isContr}(\Pi(a : A), P(a)).$$

Proof. (F1 ⇒ F2): If $\mathsf{isEquiv}(\mathsf{happly}_{f,g})$, then we have a map $f \sim g \to f = g$, by definition.

(F2 ⇒ F3): Let $P : A \to \mathcal{U}$ be a family of propositions, and let $f, g : \Pi(a : A), P(a)$. We wish to see $f = g$. By assumption we have $f \sim g \to f = g$, so it is enough to show $\Pi(a : A), f(a) = g(a)$, but we know that $P(a)$ is a proposition for all a.

(F3 ⇒ F4): Let $P : A \to \mathcal{U}$ be a family of contractible types. Then we have some function $f : \Pi(a : A), P(a)$ by taking the center of contraction of each $P(a)$. Moreover, we have that each $P(a)$ is a proposition, and so by assumption, $\Pi(a : A), P(a)$ is a proposition. As a proposition with an element, $\Pi(a : A), P(a)$ is contractible.

(F4 ⇒ F1): Let $B : A \to \mathcal{U}$ and fix $f : \Pi(x : A), B(x)$. We need to see

$$\Pi(g : \Pi(x : A), B(x)), \mathsf{isEquiv}(\mathsf{happly}_{f,g}),$$

By Theorem 1.28, this happens iff the map

$$\lambda(g, p).\, \mathsf{happly}_{f,g}(p) : \Big(\Sigma(g : \Pi(x : A), B(x)), f = g\Big) \to \Big(\Sigma(g : \Pi(x : A), B(x)), f \sim g\Big)$$

is an equivalence. As the type $\Sigma(g : \Pi(x : A), B(x)), f = g$ is contractible, it suffices to show that

$$\Sigma(g : \Pi(x : A), B(x)), f \sim g$$

is contractible. We do this by showing that it is a retract of the type

$$\Pi(x : A), \Sigma(b : B(x)), f(x) = b,$$

which is contractible by the assumption F4.

Define

$$H : \Big(\Sigma(g : \Pi(x : A), B(x)), \Pi(x : A), f(x) = g(x)\Big) \to \Pi(x : A), \Sigma(b : B(x)), f(x) = b$$
$$H(g, p) \overset{\mathsf{def}}{=} \lambda x.(g(x), p(x))$$

and

$$J : \Big(\Pi(x : A), \Sigma(b : B(x)), f(x) = b\Big) \to \Sigma(g : \Pi(x : A), B(x)), \Pi(x : A), f(x) = g(x)$$
$$J(p) \overset{\mathsf{def}}{=} (\lambda x.\, \mathsf{pr}_0(p(x)), \lambda x.\, \mathsf{pr}_1(p(x)))$$

Then we have

$$J(H(g, p)) = J(\lambda x.(g(x), p(x))) = (\lambda x.g(x), \lambda x.p(x)) = (g, p),$$

so that J is a left inverse of H. $\square$

We wish to posit a form of function extensionality that we can show to be a proposition without building too much theory. We show that F3 above is a proposition as follows: We show using F3 and F1 that the property of being a proposition is a proposition. We then use F3 again to show that F3 is a proposition. By Lemma 2.2 below, it turns out that as a result we can show F3 to be a proposition without the additional assumptions. To aid readability, let us call a class $T : \mathcal{U} \to \mathcal{U}$ of types an *exponential ideal in* $\mathcal{U}$ when for any $B : A \to \mathcal{U}$ we have

$$\big(\Pi(x : A), T(B(x))\big) \to T\big(\Pi(x : A), B(x)\big),$$

so that F3 says that propositions are an exponential ideal, and F4 says that singletons are an exponential ideal. Note that we are generalizing the usual definition of exponential ideal to dependent functions.

In order to show that being a proposition is a proposition, we need two results. First, we show that if A can be shown to be a proposition under the assumption that A has an element, then A is already a proposition.

Lemma 2.2. *Let A be a type with a function $A \to \mathsf{isProp}(A)$. Then A is a proposition.*

Proof. Let $f : A \to \mathsf{isProp}(A)$, and let $x, y : A$. We have $f(x) : \Pi(y, z : A), y = z$, so then $f(x)(x, y) : x = y$. $\square$

Second, we show that propositions are (-1)-types; from this, it follows that propositions are sets.

Lemma 2.3. *A type P is a proposition iff it is a (-1)-type. That is, P is a proposition iff all path types in P are contractible.*

Proof. Let P be a (-1)-type, so that $\Pi(x, y : P), \mathsf{isContr}(x = y)$. Then taking the center of contraction of $x = y$ gives us $\Pi(x, y : P), x = y$.

Conversely, let $w : \mathsf{isProp}(P)$, and fix $x : P$. Then $u \overset{\mathsf{def}}{=} \lambda y.w(x, y)$ has type $\Pi(y : P), x = y$. Given any $p : y = z$, we have

$$\mathsf{apd}_u(p) : \mathsf{transport}^{\lambda y.x=y}(p, u(y)) = u(z).$$

By Theorem 1.19, this gives us $u(y) \cdot p = u(z)$, and so

$$p = u(y)^{-1} \cdot u(z),$$

but p is arbitrary. Then for $z \equiv x$, and any $p, q : y = x$ we have

$$p = u(y)^{-1} \cdot u(x) = q,$$

and so $x = y$ is a proposition. As we also have $w(x, y) : x = y$, we have that $x = y$ is contractible. $\square$

Corollary 2.4. *If P is a proposition, then P is a set.*

Proof. As P is a proposition, its path types are contractible, and so path types in P are propositions. $\square$

Theorem 2.5. *If propositions form an exponential ideal in $\mathcal{U}$, then for any type $X : \mathcal{U}$, the type $\mathsf{isProp}(X)$ is a proposition.*

Proof. By Lemma 2.2, it is enough to show

$$\mathsf{isProp}(X) \to \mathsf{isProp}(\mathsf{isProp}(X)).$$

Let $i : \mathsf{isProp}(X)$, so that for all $x, y : X$, we have that $x = y$ is also a proposition by Lemma 2.3. Then $\Pi(x, y : X), x = y$ is a product of propositions, and then $\Pi(x, y : X), x = y$ is itself a proposition by assumption. □

By a similar argument we have

Theorem 2.6. *If propositions form an exponential ideal in $\mathcal{U}$, then for any X, we have*

$$\mathsf{isProp}(\mathsf{isContr}(X)).$$

Proof. Let $(c, w)(c', w') : \mathsf{isContr}(x)$. Then we have $c = c'$ by some path $p = w(c')$. Moreover, since $\mathsf{isContr}(X)$ implies $\mathsf{isProp}(X)$, we have that $\Pi(x : X), c' = x$ is a proposition, so w transports over p to w'. □

Theorem 2.7. *If propositions form an exponential ideal in $\mathcal{U}$, then for any $A : \mathcal{U}$ and any $P : A \to \mathcal{U}$, the type*

$$\big(\Pi(a : A), \mathsf{isProp}(P(a))\big) \to \mathsf{isProp}(\Pi(a : A), P(a))$$

is a proposition.

Proof. Since propositions form an exponential ideal, the type $\mathsf{isProp}(X)$ is a proposition for any type X. Also because propositions form an exponential ideal, if Y is a proposition, then $B \to Y$ is a proposition for any type B. Then in particular, we have that

$$\big(\Pi(a : A), \mathsf{isProp}(P(a))\big) \to \mathsf{isProp}(\Pi(a : A), P(a))$$

is a proposition. □

By Lemma 2.2 we have the following corollary.

Corollary 2.8. *The type*

$$\big(\Pi(a : A), \mathsf{isProp}(P(a))\big) \to \mathsf{isProp}(\Pi(a : A), P(a))$$

is a proposition. I.e., the statement that propositions form an exponential ideal is a proposition.

Now that we know that whether propositions form an exponential ideal is a proposition, we may assert this as an axiom. Here instead, we take it as a property of a universe

Definition 2.9 (Function extensionality)**.** A universe $\mathcal{U}$ *satisfies function extensionality* when propositions form an exponential ideal in $\mathcal{U}$. That is, the type

$$\mathsf{ExpProp}_{\mathcal{U}} \stackrel{\mathsf{def}}{=} \Pi(A : \mathcal{U}), \Pi(P : A \to \mathcal{U}), \big(\Pi(a : A), \mathsf{isProp}(P(a))\big) \to \mathsf{isProp}(\Pi(a : A), P(a))$$

has an element.

The axiom of function extensionality is usually given via statements F1 or F2 of Theorem 2.1, although Voevodsky preferred F4. We are most interested in statement F1. Let $\mathsf{FunExt} : \mathcal{U}_1$ be the type

$$\mathsf{FunExt} \stackrel{\mathsf{def}}{=} \Pi(A : \mathcal{U}), \Pi(B : A \to \mathcal{U}), \Pi(f, g : \Pi(x : A), P(x)), \mathsf{isEquiv}(\mathsf{happly}_{f,g}),$$

so that function extensionality gives an element of FunExt. From this for any $f, g : \Pi(x : A), B(x)$ we get a map

$$\mathsf{funext}_{f,g} : (f \sim g) \to (f = g),$$

which is an inverse to $\mathsf{happly}_{f,g}$. We will see shortly that $\mathsf{isEquiv}(f)$ is a proposition for all f, and so FunExt is itself a proposition, but already we know that the type $\mathsf{ExpProp}$ is a retract of FunExt.

Function extensionality is the first extension to MLTT we need to develop the univalent approach. We will assume function extensionality in Part II, and often use it tacitly—showing that two functions are equal by showing a homotopy between them. However, we will not assume function extensionality just yet, as we show in Section 2.4 that function extensionality follows from the univalence axiom. The other extensions we need are *truncations* and *proposition extensionality*, which we cover shortly.

2.2 Logic of propositions

Recall that logic is traditionally interpreted in type theory according to the Curry-Howard interpretation,

$$
\begin{aligned}
P \wedge Q &\equiv P \times Q \\
P \vee Q &\equiv P + Q \\
P \Rightarrow Q &\equiv P \to Q \\
\neg P &\equiv P \to \emptyset \\
\forall (a : A), P(a) &\equiv \Pi (a : A), P(a) \\
\exists (a : A), P(a) &\equiv \Sigma (a : A), P(a).
\end{aligned}
$$

This interpretation is not always well-behaved. Consider, for example, the image of a function $f : A \to B$, which is the set of all $b : B$ such that there exists $a : A$ with $f(a) = b$. Using Σ for exists, we have the *Curry-Howard image*, $\mathsf{im}_{\mathrm{CH}}(f) : \mathcal{U}$ as the type

$$\mathsf{im}_{\mathrm{CH}}(f) \overset{\mathsf{def}}{=} \Sigma (b : B), \Sigma (a : A), f(a) = b.$$

However, this type is equivalent to A:

Lemma 2.10. *For any $f : A \to B$, the second projection $\mathsf{pr}_1 : \mathsf{im}_{\mathrm{CH}}(f) \to A$ is an equivalence, and f factors as $f(x) = \mathsf{pr}_0(\mathsf{pr}_1^{-1}(x))$.*

Proof. Reshuffling the quantifiers in $\mathsf{im}_{\mathrm{CH}}(f)$ gives an equivalence

$$\Big(\Sigma (b : B), \Sigma (a : A), f(a) = b\Big) \simeq \Big(\Sigma (a : A), \Sigma (b : B), f(a) = b\Big).$$

As $\Sigma (b : B), f(a) = b$ is the singleton at $f(a)$ we have

$$\Big(\Sigma (a : A), \Sigma (b : B), f(a) = b\Big) \simeq \big(\Sigma (a : A), 1\big),$$

and the first projection gives an equivalence. The second projection $\mathsf{pr}_0\, \mathsf{im}_{CH}(f) \to A$ is the composition of these two maps, and so is itself an equivalence. An inverse is then given by the map $e(a) = (f(a), \mathsf{refl})$. Then for any $a : A$ we have that $f(a) = \mathsf{pr}_0(e(a))$. $\square$

The point is that $\Sigma(x : A), P(x)$ is not guaranteed to be a proposition, even if $P(x)$ is a proposition for each $x : A$, so using Σ carries extra structure. In this case, the extra structure allows us to recover all of A. Instead, we want a *logic of propositions* in which $\exists(x : A), P(x)$ is guaranteed to be a proposition, so that to say that the statement b is in the image of f corresponds to a propositional version of saying that the fiber of b is inhabited. Then we can define the image to be

$$\mathsf{im}(f) \equiv \Sigma(b : B), \exists(a : A), f(a) = b.$$

We immediately see that we cannot in general expect an equivalence between $\mathsf{im}(f)$ and the type

$$\exists(a : A), \Sigma(b : B), f(a) = b,$$

as this type is a proposition, but the image need not be.

Topos logic resolves this issue by restricting our attention to subsingletons. We can instead use isProp in Section 1.5. However, in order to encode a proposition-valued version of $\exists$, or even a proposition-valued version of $\vee$, we must extend our type theory with *propositional truncation*.

Definition 2.11. For any type $X : \mathcal{U}$, a *propositional truncation* of X is a type $\|X\|$ which is a proposition, together with a map

$$|-| : X \to \|X\|$$

such that if P is any other proposition, any map $f : X \to P$ factors uniquely through the map $|-|$.

We say that a type X *is inhabited* when $\|X\|$ has an element.

The truncation can be defined in several ways: by directly extending MLTT; as a higher inductive type (Section 2.7); via resizing (Section 2.5), or by assuming the law of excluded middle.

In particular, if we assume the law of excluded middle: we have $A \to (\neg\neg A)$ for any type A. Now take any proposition P such that $A \to P$. By contraposition we have $\neg P \to \neg A$, and then since both $\neg A$ and $\neg P$ are propositions, using excluded middle we can get $\neg\neg A \to \neg\neg P$. As P is a proposition, we may (again using excluded middle) eliminate the double negation in front of P to get $\neg\neg A \to P$.

The higher-inductive type we give in Section 2.7 directly expresses the universal property of truncation as a rule of construction, while the resizing and LEM approach encode the propositional truncation concretely. Formally, in the absence of a general framework for higher-inductive types (see the discussion in Section 2.8), the higher-inductive type must be added directly, so the higher-inductive approach corresponds to adding truncations directly. This was already studied as *squash types* before univalent type theory arose.

As we are working informally, it is enough for us to say that we are positing directly that each type X has a propositional truncation. Once we have the truncation, we can begin to study how it behaves. In particular, truncation is idempotent up to equivalence, which follows from a more general observation.

Lemma 2.12. *If P is a proposition, then $P \simeq \|P\|$.*

Proof. We need only see that $P \Leftrightarrow \|P\|$. Since P is a proposition, we have that $\mathsf{id} : P \to P$ factors through the map $|-| : P \to \|P\|$. □

The utility of the above lemma is as follows: We will often define some operation T' on types such that $T'(X) = \|T(X)\|$. The type $T(X)$ may contain useful information, but truncation hides this. Often, however, $T(X)$ is already a proposition, so we have that $T'(X) \simeq T(X)$ so we can extract the information from $T(X)$ already by knowing $\|T(X)\|$. More generally, we have

Lemma 2.13. *If P is a proposition and $P \Leftrightarrow X$, then $P \simeq \|X\|$.*

Proof. As $X \to P$ and P is a proposition, we have $\|X\| \to P$. Moreover, we have the composite map $P \to X \to \|X\|$. Hence, $P \Leftrightarrow \|X\|$. □

Lemma 2.14. *Truncation is functorial: for any types X and Y,*

$$(X \to Y) \to (\|X\| \to \|Y\|).$$

Proof. Suppose $f : X \to Y$. Then we have $\lambda x.|f(x)| : X \to \|Y\|$. As $\|Y\|$ is a proposition, we then have a map $\|X\| \to \|Y\|$. $\square$

By truncating the Curry-Howard interpretation everywhere that it fails to give a proposition, we get an interpretation of logic at the level of propositions.

$$\begin{aligned}
P \wedge Q &\overset{\mathsf{def}}{=} P \times Q \\
P \vee Q &\overset{\mathsf{def}}{=} \|P + Q\| \\
P \Rightarrow Q &\overset{\mathsf{def}}{=} P \to Q \\
\neg P &\overset{\mathsf{def}}{=} P \to \emptyset \\
\forall(a : A), P(a) &\overset{\mathsf{def}}{=} \Pi(a : A), P(a) \\
\exists(a : A), P(a) &\overset{\mathsf{def}}{=} \|\Sigma(a : A), P(a)\|
\end{aligned}$$

Where P and Q are propositions on the first three lines, and $P : A \to \mathcal{U}$ is a predicate on a type A in the last two lines. Each of these types is a proposition: The type $P \wedge Q$ is a proposition because P and Q are propositions; the types $P \vee Q$ and $\exists(a : A), P(a)$ are propositions by definition, while $P \to Q$ and $\Pi(a : A), P(a)$ are propositions by function extensionality. In general, we will only use the logical notation just defined when we are using $\exists$ or $\vee$, or trying to make explicit the connection between what we are doing and some traditional logical principle.

This interpretation allows us to define the image as we wanted to above.

Definition 2.15. The *image* of f is the type

$$\mathsf{im}(f) \overset{\mathsf{def}}{=} \Sigma(b : B), \|\mathsf{fib}_f(b)\|.$$

That is,

$$\mathsf{im}(f) \overset{\mathsf{def}}{=} \Sigma(b : B), \exists(a : A), (f(a) = b).$$

That is, the image of f is the type of all $b : B$ such that there exists (as property) an $a : A$ with $f(a) = b$.

While in general $+$ is not guaranteed to give propositions, the coproduct of disjoint propositions is always again a proposition.

Lemma 2.16. *If A and B are disjoint propositions (I.e., such that $\neg(A \times B)$), then $A+B$ is a proposition; hence $A + B \simeq A \vee B$.*

Proof. Let $x, x' : A + B$, and do a case analysis on x and x':

- Case 1: $x = \mathsf{inl}\, a$ and $x' = \mathsf{inl}\, a'$. As A is a proposition, $a = a'$, and so $\mathsf{inl}\, a = \mathsf{inl}\, a'$.
- Case 2: $x = \mathsf{inr}\, b$ and $x' = \mathsf{inr}\, b'$. Similar to Case 1.
- Case 3: $x = \mathsf{inl}\, a$ and $x' = \mathsf{inr}\, b'$. Then we have $(a, b') : A \times B$, but we assumed $\neg(A \times B)$, and so we get $x = x'$.
- Case 4: $x = \mathsf{inr}\, b$ and $x' = \mathsf{inl}\, a'$ Similar to Case 3.

□

Definition 2.17. A type P is *decidable* if $P + \neg P$.
A predicate $P : A \to \mathcal{U}$ is *decidable* if $\Pi(a : A), P(a) + \neg P(a)$.
A type A *has decidable equality* or *is discrete* when for all $a, a' : A$, the type $a = a'$ is decidable.

This terminology is standard in constructive mathematics, but since we also deal with computability theory, this terminology creates a clash. We will always use *recursive* or *computable* when discussing recursive decidability; nevertheless, to avoid confusion we will sometimes say that P is *complemented* when $P + \neg P$. The standard example of a discrete type is the type of natural numbers, which is decidable by the same argument as Theorem 1.13.

Lemma 2.18. *For any family of types $P : X \to U$, we have*

$$\big\| \Sigma(x : X), \|P(x)\| \big\| \simeq \|\Sigma(x : X), P(x)\|$$

Proof. We need only show logical equivalence. For the implication

$$\big\| \Sigma(x : X), \|P(x)\| \big\| \to \|\Sigma(x : X), P(x)\| ,$$

it is enough to define a map

$$\big(\Sigma(x : X), \|P(x)\| \big) \to \|\Sigma(x : X), P(x)\| ,$$

and by currying, such a map is the same thing as a map

$$\Pi(x : X), \|P(x)\| \to \|\Sigma(x : X), P(x)\| .$$

Fixing $x : X$, a map $\|P(x)\| \to \|\Sigma(x : X), P(x)\|$ arises by functoriality of truncation from any map $P(x) \to \Sigma(x : X), P(x)$, and $\lambda p.(x, p)$ suffices.

For the other direction, it is enough to define a map

$$\big(\Sigma(x : X), P(x)\big) \to \Sigma(x : X), \|P(x)\| ,$$

and $\lambda(x, p).(x, |p|)$ suffices. □

Lemma 2.19. *If X is a type and $P : X \to \mathcal{U}$ is a family of types over X such that* $\mathsf{isProp}\,(\Sigma(x : X), \|P(x)\|)$, *then* $(\Sigma(x : X), \|P(x)\|) \simeq \|\Sigma(x : X), P(x)\|$.

Proof. As $\mathsf{isProp}\,(\Sigma(x : X), \|P(x)\|)$, we have an equivalence

$$\|\Sigma(x : X), \|P(x)\|\| \simeq \Sigma(x : X), \|P(x)\| ,$$

and the latter is equivalent to $\|\Sigma(x : X), P(x)\|$ by the previous lemma. □

2.3 Surjections, embeddings and equivalences

We can decompose $\mathsf{isEquiv}(f)$ into $\mathsf{linv}(f) \times \mathsf{rinv}(f)$, which correspond to saying that f is a section and f is a retraction. We can similarly decompose the notion of having contractible fibers: since $\mathsf{isContr}(A)$ is equivalent to $A \times \mathsf{isProp}(A)$, for $f : A \to B$, we can restate $\mathsf{isContr}(f)$ as

$$\Pi(b : B), \mathsf{isProp}(\mathsf{fib}_f\, b) \times \|\mathsf{fib}_f(b)\| .$$

We can then split these two notions to arrive at the following definitions.

Definition 2.20. A function $f : A \to B$ is

- an *embedding* if it has propositional fibers:

$$\Pi(b : B), \mathsf{isProp}(\mathsf{fib}_f\, b).$$

- a *surjection* if it has inhabited fibers:

$$\Pi(b : B), \|\mathsf{fib}_f(b)\|.$$

Then this directly gives us that a function has contractible fibers iff it is both a surjection and an embedding. By Theorem 1.21, we have that a function is an equivalence iff it is both an embedding and a surjection.

Note that we follow the HoTT Book in highlighting the subtlety of the definition by using the word *embedding* rather than *injection*. However, the HoTT Book defines embedding by looking at ap_f, which is sometimes more useful. Our definition instead gives a symmetry between embedding and surjection. Nevertheless, the above definition is indeed equivalent to the definition given in the HoTT Book.

Lemma 2.21. *A function $f : A \to B$ is an embedding iff $\mathsf{ap}_f : (x = y) \to (f(x) = f(y))$ is an equivalence for each $x, y : A$.*

Proof. We need to see that ap_f has contractible fibers iff f has propositional fibers. Fix $b : B$, and $(x, p), (y, q) : \mathsf{fib}_f(b)$. We have

$$\begin{aligned}\big((x,p) = (y,q)\big) &\simeq \Sigma(r : x = y), p = \mathsf{ap}_f(r) \cdot q \\ &\simeq \Sigma(r : x = y), \mathsf{ap}_f(r) = p \cdot q^{-1} \\ &\simeq \mathsf{fib}_{\mathsf{ap}_f}(p \cdot q^{-1}).\end{aligned}$$

So if ap_f has contractible fibers, then f has propositional fibers. On the other hand, for any $p : f(x) = f(y)$ we have (x, p) and (y, refl) in $\mathsf{fib}_f(f(y))$, so that $\big((x,p) = (y, \mathsf{refl})\big) \simeq \mathsf{fib}_{\mathsf{ap}_f}(p)$. Then if f has propositional fibers, ap_f has contractible fibers. $\square$

In the case where B is a set, we can simplify the above characterization; in this case, it is enough to know that there is a map $f(x) = f(y) \to x = y$. We will give a somewhat indirect proof, in order to introduce an important observation by Martín Escardó concerning retracts of identity types.

Lemma 2.22. *Suppose $R : X \to X \to \mathcal{U}$ and that we have maps*

$$r : \Pi(x, y : X), x = y \to R(x, y)$$
$$s : \Pi(x, y : X), R(x, y) \to x = y,$$

such that $r_{x,y}$ is a left inverse of $s_{x,y}$ for all $x, y : X$. Then $r_{x,y}$ and $s_{x,y}$ are inverse. Hence, there is an equivalence $R(x, y) \simeq (x = y)$ for all $x, y : X$.

Proof. Since we already have that $r_{x,y} \circ s_{x,y}$ is homotopic to the identity, then we only need to see that $s(r(p)) = p$ for all $p : x = y$. Notice that $s \circ r$ is an idempotent function of type $\Pi(x, y : X), (x = y) \to (x = y)$, since r is left inverse to s. Then it is enough to show that any family of functions

$$f : \Pi(x, y : X), (x = y) \to (x = y)$$

such that $(f \circ f) \sim f$ is homotopic to the identity. Since refl is the identity with respect to composition, we have

$$f(\mathsf{refl}) = f(\mathsf{refl}) \cdot \mathsf{refl}$$

for any function $f : \Pi(x, y : X), (x = y) \to (x = y)$. Then by path induction, any function $f : \Pi(x, y : X), (x = y) \to (x = y)$ satisfies

$$\Pi(x, y : X), \Pi(p : x = y), f(p) = f(\mathsf{refl}_x) \cdot p.$$

In particular, if f is idempotent we have

$$f(p) = f(f(p)) = f(\mathsf{refl}) \cdot f(p),$$

where the second equality is the previous observation. Then composition with $f(p)^{-1}$, gives $f(\mathsf{refl}) = \mathsf{refl}$. Applying path induction, we then have that for any $p : x = y$ that $f(p) = p$ for any family of idempotent functions on $x = y$. In particular, we have that $(s \circ r)(p) = p$, so s is also a left inverse of r. $\square$

Theorem 2.23. *If B is a set, then $f : A \to B$ is an embedding iff for each $x, y : A$ there is a map $h : (f(x) = f(y)) \to (x = y)$.*

Proof. As B is a set, $f(x) = f(y)$ is a proposition, so h tells us that $f(x) = f(y)$ is a retract of $x = y$. So, for every $x, y : A$ we have $f(x) = f(y)$ is a retract of $x = y$, and by Lemma 2.22 we have an equivalence

$$(f(x) = f(y)) \simeq (x = y). \qquad \square$$

We have stated embedding and surjection as if they were property, not structure. Function extensionality tells us that this is justified:

Lemma 2.24. *In the presence of function extensionality, being an embedding, being a surjection and having contractible fibers are all propositions.*

Proof. Function extensionality tells us that propositions form an exponential ideal and that $\mathsf{isProp}(X)$ is always a proposition, so being an embedding is a proposition; similarly since $\mathsf{isContr}(X)$ is a proposition for any $X : \mathcal{U}$, so is $\mathsf{isContr}(f)$ for any $f : A \to B$; again using function extensionality, since truncations are propositions by definition, we have that being a surjection is a proposition. $\square$

We have alluded to the fact that $\mathsf{isContr}(f)$ and $\mathsf{isEquiv}(f)$ are equivalent. In the presence of function extensionality, it is enough to show that $\mathsf{isEquiv}(f)$ is a proposition, since $\mathsf{isContr}(f)$ and $\mathsf{isEquiv}(f)$ are logically equivalent. First note that if $f : A \to B$ has an inverse, then the composition maps $(f \circ -) : (C \to A) \to (C \to B)$ and $(- \circ f) : (B \to C) \to (A \to C)$ do as well, by composition with the inverse of f. As a result, we have the following.

Theorem 2.25. *If $\mathcal{U}$ satisfies function extensionality, then for any $A, B : \mathcal{U}$, if $f : A \to B$ has an inverse, then $\mathsf{rinv}(f)$ and $\mathsf{linv}(f)$ are contractible.*

Proof. Fix $f : A \to B$ with an inverse. By function extensionality, $\mathsf{linv}(f)$ is equivalent to the fiber of $(- \circ f)$ over id_A. By the above observation, we know that $(- \circ f)$ has an inverse, and so has contractible fibers. Similarly, $\mathsf{rinv}(f)$ is equivalent to the fiber of $(f \circ -)$ over id_B. $\square$

Theorem 2.26. *If $\mathcal{U}$ satisfies function extensionality, then for any $A, B : \mathcal{U}$ and any $f : A \to B$, we have that $\mathsf{isEquiv}(f)$ is a proposition.*

Proof. Note that $\mathsf{isEquiv}(f) \simeq \mathsf{linv}(f) \times \mathsf{rinv}(f)$. If $e : \mathsf{isEquiv}(f)$, then f is invertible, so $\mathsf{isEquiv}(f)$ is a product of contractible types by function extensionality, and so is a proposition. Briefly, we have

$$\mathsf{isEquiv}(f) \to \mathsf{isProp}(\mathsf{isEquiv}(f)).$$

Now apply Lemma 2.2. □

Corollary 2.27. *If $\mathcal{U}$ satisfies function extensionality, then for any $A, B : \mathcal{U}$ and any $f : A \to B$ we have*

$$\mathsf{isContr}(f) \simeq \mathsf{isEquiv}(f).$$

Proof. Function extensionality tells us that the types are logically equivalent propositions. □

2.4 Proposition extensionality and univalence

As with functions, MLTT provides no way to show that two *types* are equal, so if we want to prove types to be equal, we need an extensionality principle for types. Before examining the situation for all types, we look at propositions.

In logical languages (including the Mitchell-Benabou language), two propositions are considered equal if they are logically equivalent. This is not possible in MLTT with our definition of proposition, so we suggest an extensionality principle for propositions.

Definition 2.28. A universe $\mathcal{U}$ *satisfies proposition extensionality* if whenever $A : \mathcal{U}$ and $B : \mathcal{U}$ are propositions and $A \Leftrightarrow B$, then $A = B$: Let

$$\mathsf{PropExt}_{\mathcal{U}} \stackrel{\mathsf{def}}{=} \Pi(A, B : \mathcal{U}), \mathsf{isProp}(A) \to \mathsf{isProp}(B) \to (A \Leftrightarrow B) \to (A = B).$$

$\mathcal{U}$ satisfies proposition extensionality when there is $\mathsf{propext} : \mathsf{PropExt}_{\mathcal{U}}$.

Notice that function extensionality tells us already that $A \simeq B$ is a proposition when A and B are propositions, since $A \simeq B$ is a proposition indexed sum of propositions. This gives us the following slight strengthenings of Lemma 1.7 and Lemma 1.11.

Lemma 2.29. *If $\mathcal{U}$ satisfies function extensionality, and $A : \mathcal{U}$ and $B : \mathcal{U}$ are propositions, then*

$$(A \Leftrightarrow B) \simeq (A \simeq B).$$

Proof. We have that

$$(A \Leftrightarrow B) \Leftrightarrow (A \simeq B),$$

and both sides are propositions, so this implies

$$(A \Leftrightarrow B) \simeq (A \simeq B). \qquad \square$$

So in the presence of function extensionality we can reformulate the type of proposition extensionality as

$$\Pi(A, B : \mathcal{U}), \mathsf{isProp}(A) \to \mathsf{isProp}(B) \to (A \simeq B) \to (A = B).$$

Since we have that $A \simeq B$ is a proposition when both types are propositions, this tells us that $A \simeq B$ is a retract of $A = B$, and since $A \simeq B$ is a proposition, the retraction can be given by any map $(A = B) \to (A \simeq B)$. We may as well use the map $\mathsf{idtoequiv}$. In short, we have that proposition extensionality and function extensionality together imply that $\mathsf{idtoequiv} : (A = B) \to (A \simeq B)$ has a right inverse when A and B are propositions. As a corollary, we have

Lemma 2.30. *Proposition extensionality and function extensionality together imply*

$$\Pi(A, B : \mathcal{U}), \mathsf{isProp}(A) \to \mathsf{isProp}(B) \to \mathsf{isEquiv}(\mathsf{idtoequiv}_{A,B}).$$

This principle is known as *propositional univalence*.

In the section on function extensionality, we said we wanted axioms to be propositions, but we have not yet shown proposition extensionality to be a proposition.

Theorem 2.31. *If $\mathcal{U}$ satisfies function extensionality, then propositional univalence implies $A = B$ is a proposition whenever A and B are propositions. Hence, in the presence of function extensionality, proposition extensionality is a proposition.*

Proof. When A and B are propositions, we already know $(A \simeq B) \simeq (A \Leftrightarrow B)$. As $A \Leftrightarrow B$ is a proposition, proposition extensionality implies that $A = B$ is a proposition as well.

As proposition extensionality implies propositional univalence, we have

$$\mathsf{PropExt} \to \mathsf{isProp}(A = B),$$

whenever A and B are propositions. The type of proposition extensionality is of the form $\Pi(x : X), A = B$, and since propositions form an exponential ideal, we have

$$\mathsf{PropExt} \to \mathsf{isProp}(\mathsf{PropExt}).$$

Hence, $\mathsf{PropExt}$ is a proposition. $\square$

In the statement of propositional univalence, we could drop the condition that A and B are propositions, to get the type

$$\mathsf{UA}_{\mathcal{U}} \overset{\mathsf{def}}{=} \Pi(A, B : \mathcal{U}), \mathsf{isEquiv}(\mathsf{idtoequiv}_{A,B}).$$

This type says that all equivalences arise from a path $A = B$. The axiom that it is inhabited is known as the *univalence axiom*.

Definition 2.32. A universe $\mathcal{U}$ is *univalent* when there is an element of $\mathsf{UA}_{\mathcal{U}}$. In particular, if $\mathcal{U}$ is univalent, for $A, B : \mathcal{U}$ we have a map

$$\mathsf{ua} : (A \simeq B) \to (A = B)$$

such that ua and $\mathsf{idtoequiv}$ are inverse.

In fact, by Lemma 2.22, it is enough to say that $\mathsf{idtoequiv}$ has a left inverse.

Theorem 2.33. *If* $\mathsf{idtoequiv} : (A = B) \to (A \simeq B)$ *has a right inverse for each* $A, B : \mathcal{U}$*, then* $\mathsf{idtoequiv}$ *is an equivalence for each* $A, B : \mathcal{U}$*.*

Function extensionality and proposition extensionality follow from univalence. Proposition extensionality follows directly, but function extensionality takes more work. The argument here is abstracted from Voevodsky's original proof that univalence implies function extensionality, as presented in [33]. It hinges around the total type of the identity relation, which we call the *diagonal* of a type.

Definition 2.34. Given a type $A : \mathcal{U}$, define the *diagonal* of A to be the type

$$\Delta A \overset{\mathsf{def}}{=} \Sigma(a, a' : A), a = a'.$$

There is then a *diagonal map* $\delta_A : A \to \Delta A$ given by

$$\delta(x) \overset{\mathsf{def}}{=} (x, x, \mathsf{refl}).$$

It is straightforward to check that the diagonal map is an equivalence. In particular, we have two candidate inverses given by pr_0 and pr_1. Both are left inverse directly since we have the equalities $\mathsf{pr}_0(\delta(x)) = \mathsf{pr}_1(\delta(x)) = x$. For the other direction, we have

$$\delta(\mathsf{pr}_0(x, y, p)) = (x, x, \mathsf{refl}_x)),$$

so we wish to see that $(x, x, \mathsf{refl}_x) =_{\Delta A} (x, y, p)$. As $\mathsf{refl} : x = x$, we need to see that we have $q : x = y$ such that $\mathsf{transport}^{x=-}(q, \mathsf{refl}_x) = p$. Of course, Theorem 1.19 determines a witness $w : \mathsf{transport}(p, \mathsf{refl}_x) = \mathsf{refl}_x \cdot p$, so the given p suffices. For pr_1, the argument is similar.

Now consider the class of maps arising from paths, the *path-induced equivalences*:

$$\mathsf{PIE}_{A,B}(f) \overset{\mathsf{def}}{=} \Sigma(p : A = B), f = \mathsf{idtofun}(p).$$

Note that this type is the Curry-Howard image of $\mathsf{idtofun}$. We have $r_A : \mathsf{PIE}_{A,A}(\mathsf{id}_A)$ given by $r_A \overset{\mathsf{def}}{=} (\mathsf{refl}_A, \mathsf{refl}_{\mathsf{id}_A})$. We can then define a version of $\mathsf{idtoequiv}$ for PIE using the function $\epsilon_{A,B} : \Pi_{p:A=B} \mathsf{PIE}_{A,B}(\mathsf{idtofun}(p))$ defined by path induction with

$$\epsilon(\mathsf{refl}_A) \overset{\mathsf{def}}{=} r_A.$$

In particular, we have $\mathsf{PIE}_{A,B}(\mathsf{idtofun}(p))$ for any $p : A = B$. Write $A \simeq_P B$ for

$$\Sigma(f : A \to B), \mathsf{PIE}_{A,B}(f).$$

Then we have

$$\mathsf{idtopie} : A = B \to A \simeq_P B$$

given by

$$\mathsf{idtopie}(p) \stackrel{\mathsf{def}}{=} (\mathsf{idtofun}(p), \epsilon(p)).$$

Note that $\mathsf{idtopie}$ is one direction of an equivalence $(A \simeq_P B) \simeq (A = B)$ given by Lemma 2.10. Take $S_{A,B} : A \simeq_P B \to A = B$ to be a section of $\mathsf{idtopie}$.

Lemma 2.35. *Let $P : \Pi(A, B : \mathcal{U}), (A \to B) \to \mathcal{U}$ such that $\Pi(A : \mathcal{U}), P(\mathsf{id}_A)$. For every $A, B : \mathcal{U}$ and $p : A = B$, the map $\mathsf{idtofun}(p)$ satisfies P. Formally,*

$$\Pi(A, B : \mathcal{U}), \Pi(p : A = B), P(\mathsf{idtofun}(p)).$$

Proof. By path induction, we need only see $P(\mathsf{idtofun}(\mathsf{refl}_A))$ for each A, and we have $P(\mathsf{id}_A)$ by assumption. □

That is, we can prove something for all path-induced equivalences by looking at the identity maps

Lemma 2.36. *Let $P : \Pi(A, B : \mathcal{U}), (A \to B) \to \mathcal{U}$ such that $\Pi(A : \mathcal{U}), P(\mathsf{id}_A)$, then every path-induced equivalence in $\mathcal{U}$ satisfies P. That is, for all $A, B : \mathcal{U}$ and $f : A \to B$, $\mathsf{PIE}(f) \to P(f)$.*

Proof. Note that every $e : A \simeq_P B$ can be written as $\mathsf{idtofun}(S(e))$, and apply the previous lemma. The key point here is that $\mathsf{idtopie}$ has a section. □

Corollary 2.37. *We have that $\mathsf{PIE}(f) \to \mathsf{isEquiv}(f)$.*

Proof. The identity map is an equivalence. □

In particular, if f is a path-induced equivalence, then f is both a section and a retraction. Ultimately, univalence tells us that equivalences are exactly path-induced equivalences.

Lemma 2.38. *If $\mathcal{U}$ is univalent, then for any $A, B : \mathcal{U}$ we have*

$$\Pi(f : A \to B), (\mathsf{isEquiv}(f) \simeq \mathsf{PIE}(f)).$$

Proof. By Theorem 1.28, we have that all maps $\mathsf{PIE}(f) \to \mathsf{isEquiv}(f)$ are equivalences iff the induced map

$$e : (\Sigma(f : A \to B), \mathsf{PIE}(f)) \to (\Sigma(f : A \to B), \mathsf{isEquiv}(f))$$

is an equivalence. The induced map e factors as the equivalence $(A \simeq_P B) \simeq (A = B)$ followed by $\mathsf{idtoequiv}$. Since univalence says that $\mathsf{idtoequiv}$ is an equivalence, it says that e is the composite of equivalences. □

However, to show that univalence implies function extensionality, it is enough to assume that the path-induced equivalences satisfy two properties that are satisfied by equivalences, namely it is enough to assume

- path-induced equivalences are closed under homotopy;
- all diagonal maps $\delta_X : X \to \Delta_X$ are path-induced equivalences.

We assume these explicitly as needed in the following lemmas.

Lemma 2.39. *Suppose path-induced equivalences are closed under homotopies. If f is a path-induced equivalence, then so is $(- \circ f)$.*

Proof. We need only see that $(- \circ f)$ is homotopic to a path-induced map when $\mathsf{PIE}(f)$. First note that for any $p : X = X'$ and $g : X' \to Y$, we have $\mathsf{transport}^{\lambda X.X \to Y}(p^{-1}, g) = g \circ \mathsf{idtofun}(p)$, by path induction: when $p \equiv \mathsf{refl}$ both sides are equal to g. So then we have that

$$\mathsf{transport}^{\lambda X.X \to Y}(S(f, e)^{-1}, g) = g \circ f,$$

and by the discussion in Section 1.7, we know that every transport map is homotopic to a path-induced function. □

Theorem 2.40. *If path-induced equivalences are closed under homotopies, and for each X, we have* $\mathsf{PIE}(\delta_X)$*, then for any $f, g : A \to B$ we have $f \sim g \to f = g$.*

Proof. Note that for any X we have $\mathsf{pr}_0 \circ \delta_X = \mathsf{pr}_1 \circ \delta_X$ by definition and the computation rule for function types. As δ_X is a path-induced equivalence, so is precomposition with δ_X, but then we also have that $- \circ \delta_X$ is a section, and so we have that $\mathsf{pr}_0 =_{\Delta(X) \to X} \mathsf{pr}_1$. Now take $f, g : A \to B$ with $\eta : f \sim g$. Then we have $h : A \to \Delta B$ given by

$$h(x) \stackrel{\mathsf{def}}{=} (f(x), g(x), \eta(x)).$$

Finally, we have

$$f = \mathsf{pr}_0 \circ h = \mathsf{pr}_1 \circ h = g.$$

□

Corollary 2.41. *If $\mathcal{U}$ is univalent, then for any $A, B : \mathcal{U}$ and any $f, g : A \to B$, we have $f \sim g \to f = g$.*

Proof. We need to see that univalence implies that path-induced equivalences are closed under homotopies and that δ_X is path-induced. As univalence implies $\mathsf{PIE}(f) \simeq \mathsf{isEquiv}(f)$, it is enough to show that equivalences are closed under homotopies, which is immediate by transitivity of homotopy, and that $\mathsf{isEquiv}(\delta_X)$, which we showed above. □

In fact, the above non-dependent version of function extensionality—which we will call *naive function extensionality*—implies full function extensionality. We proceed in two steps: We first show that non-dependent function extensionality implies that $f \circ -$ is an equivalence whenever f is, and then that this implies that contractible types form an exponential ideal.

Lemma 2.42. *Fix types $A, B, X : \mathcal{U}$ and let $f : A \to B$ have inverse $g : B \to A$. If $\mathcal{U}$ satisfies naive function extensionality, then $g \circ -$ is an inverse of $f \circ - : (X \to A) \to (X \to B)$.*

Proof. Let $h : X \to A$ and $x : X$. We have that $g(f(h(x))) = h(x)$ since g and f are inverse. Then we have by naive function extensionality that $g \circ f \circ h = h$. Hence, $(g \circ -) \circ (f \circ -) \simeq \mathsf{id}_{X \to A}$. The other direction is similar. □

Lemma 2.43. *If $B : A \to \mathcal{U}$ and each $B(a)$ is contractible, then $\Pi(x : A), B(x)$ is a retract of* $\Sigma(h : A \to \Sigma(x : A), B(x)), \mathsf{pr}_0 \circ h = \mathsf{id}$.

Proof. The section map is given by

$$s(f) = (\lambda a.(a, f(a)), \mathsf{refl}),$$

while the retraction map is given by

$$r(g, p) = \lambda a.\, \mathsf{transport}^B(\mathsf{happly}(p, a), \mathsf{pr}_1(g(a))).$$

For $f : \Pi(x : A), B(x)$ we compute

$$\begin{aligned} r(s(f)) &= \lambda a.\, \mathsf{transport}^B(\mathsf{happly}(\mathsf{refl}, a)), \mathsf{pr}_1(a, f(a)) \\ &= \lambda a.\, \mathsf{transport}^B(\mathsf{refl}_a, f(a)) \\ &= \lambda a. f(a) \\ &= f. \end{aligned}$$

□

Theorem 2.44. *If $\mathcal{U}$ satisfies naive function extensionality, then $\mathcal{U}$ satisfies function extensionality.*

Proof. Let $B : A \to \mathcal{U}$ such that $B(a)$ is contractible for each $a : A$. We wish to see that $\Pi(x : A), B(x)$ is contractible. By Lemma 2.43, we know that this type is a retract of the type $\Sigma(h : A \to \Sigma(x : A), B(x)), \mathsf{pr}_0 \circ h = \mathsf{id}$; since contractible types are closed under retracts, it suffices to show that $\Sigma(h : A \to \Sigma(x : A), B(x)), \mathsf{pr}_0 \circ h = \mathsf{id}$ is contractible. Notice that this type is the fiber of id under the map

$$\mathsf{pr}_0 \circ - : (A \to \Sigma(x : A), B(x)) \to (A \to A).$$

A map is an equivalence if and only if all its fibers are contractible, so it suffices to prove that $\mathsf{pr}_0 \circ -$ is an equivalence and so by Lemma 2.42, it is enough to show that the first projection $\mathsf{pr}_0 : \Sigma(x : A), B(x) \to A$ is an equivalence. A candidate inverse for pr_0 is given by $g(a) = (a, c)$ where c is the center of contraction of $B(a)$. Then $\mathsf{pr}_0(g(a)) = a$ by refl, and then for any pair $(a, b) : \Sigma(x : A), B(x)$ we have that $(a, b) = (a, c)$, since contractible types are propositions. □

Corollary 2.45. *Any univalent universe satisfies function extensionality.*

Proof. We know that any univalent universe satisfies naive function extensionality, and that any universe satisfying naive function extensionality satisfies function extensionality. □

In fact, the converse of Lemma 2.38 holds. That is, univalence says exactly that the structure of a path-induced equivalence is the same as the structure of an equivalence. Univalence is sometimes stated informally as "all equivalences arise as transport", or "all equivalences arise from paths". We can in fact prove univalence to be equivalent to a very weak interpretation of this informal statement.

Theorem 2.46. *A universe $\mathcal{U}$ is univalent if and only if, for all $A, B : \mathcal{U}$ and $f : A \to B$ we have the implication*

$$\mathsf{isEquiv}(f) \to \mathsf{PIE}(f).$$

Proof. Given a family of maps $\eta : \mathsf{isEquiv}(f) \to \mathsf{PIE}(f)$, we have function extensionality, by Theorem 2.40. Fix some $f : A \to B$. Then, by function extensionality we have that $\mathsf{isEquiv}(f)$ is a proposition. Our assumption tells us that $\mathsf{isEquiv}(f) \Leftrightarrow \mathsf{PIE}(f)$, and since $\mathsf{isEquiv}(f)$ is a proposition, we know that any map $\mathsf{PIE}(f) \to \mathsf{isEquiv}(f)$ is a retraction with section η. In particular, the map ϵ given by path induction in Corollary 2.37 has η as a section. This then lifts to a section-retraction pair

$$\begin{aligned} s &: \big(\Sigma(f : A \to B), \mathsf{isEquiv}(f)\big) \to \big(\Sigma(f : A \to B), \mathsf{PIE}(f)\big) \\ s(f, e) &= (f, \eta(e)) \\ r &: \big(\Sigma(f : A \to B), \mathsf{PIE}(f)\big) \to \big(\Sigma(f : A \to B), \mathsf{isEquiv}(f)\big) \\ r(f, e) &= (f, \epsilon(e)) \end{aligned}$$

By Lemma 2.22, we only need to see that $\mathsf{idtoequiv}$ has a section, but we defined ϵ so that $\mathsf{idtoequiv}(p) \sim (\mathsf{idtofun}(p), \epsilon(\mathsf{idtofun}(p)))$, which means that $\mathsf{idtoequiv}$ factors as $r \circ \mathsf{idtopie}$. As the projection $\mathsf{pr}_= : (A \simeq_P B) \to (A = B)$ is a section of $\mathsf{idtopie}$, the composite $\mathsf{pr}_= \circ s$ is a section of $\mathsf{idtoequiv}$. □

We are finally in a place to show that univalence is a proposition.

Theorem 2.47. *The type $\mathsf{UA}_{\mathcal{U}}$, is a proposition.*

Proof. By Lemma 2.2 it is enough to show that univalence implies that $\mathsf{UA}_{\mathcal{U}}$ is a proposition. We know that univalence implies function extensionality. Since function extensionality implies that $\mathsf{isEquiv}(f)$ is a proposition and that propositions form an exponential ideal, function extensionality implies $\mathsf{isProp}(\mathsf{UA}_{\mathcal{U}})$. Then univalence implies $\mathsf{isProp}(\mathsf{UA}_{\mathcal{U}})$. □

The use of equivalences instead of functions with an inverse in the statement of univalence is crucial. Most of the arguments in this section work when we replace $\mathsf{isEquiv}(f)$ with $\mathsf{inverse}(f)$ everywhere. In particular, If we replace $\mathsf{idtoequiv}$ with the corresponding function

$$i : A = B \to \Sigma(f : A \to B), \mathsf{inverse}(f),$$

then the corresponding version of univalence tells us that $\mathsf{inverse}(f)$ is a section of $\mathsf{PIE}(f)$, and so $\mathsf{inverse}(f)$ must be a proposition. However, we mentioned before that if $f : A \to B$ has an inverse, then

$$\mathsf{inverse}(f) \simeq \Pi(x : A), x = x.$$

So, if A is not a set, then $\mathsf{inverse}(f)$ is not a proposition. Moreover, both univalence and the corresponding statement with $\mathsf{inverse}(f)$ imply that there are types which are not sets. Consequently, the type

$$\Pi(A, B : \mathcal{U}), (A = B) \simeq \big(\Sigma(f : A \to B), \mathsf{inverse}(f)\big),$$

is not inhabited.

We will limit our use of univalence in the main development, using instead function extensionality and proposition extensionality. The reason is twofold. First, since we work primarily at the level of sets and propositions, it is usually enough to use these two weaker principles; the full power of univalence is more than we require. Secondly, the discussion of topos logic above is not simply for comparison; while proposition extensionality as stated in this section is not expressible in topos logic, a form of proposition extensionality does hold. As a result, the technical work in Part II should translate with a little work to a topos-theoretic setting, as long as we do not use univalence. In other words, we hope that the material in Part II will still have technically interesting content to a mathematician who rejects or is otherwise uninterested in univalent mathematics.

We will use function extensionality and proposition extensionality for all universes throughout. This means in particular that the following all hold in all universe.

- propositions form an exponential ideal (Theorem 2.1.F3);
- contractible types form an exponential ideal (Theorem 2.1.F4);
- all contractible types are equal (Corollary 1.12);
- propositional univalence (Lemma 2.30);
- proposition extensionality is a proposition (Theorem 2.31);
- for any function $f : A \to B$, we have that $\mathsf{isEquiv}(f)$ is a proposition (Theorem 2.26);
- For any $f : A \to B$, the types $\mathsf{isEquiv}(f)$ and $\mathsf{isContr}(f)$ are equivalent (Corollary 2.27).

Proposition extensionality and function extensionality allow us to strengthen Lemma 1.11.

Lemma 2.48. *The following are equivalent (in fact, equal) for any type A:*

- $A \simeq 1$; *(A is equivalent to 1)*
- $\mathsf{isContr}(A)$; *(A is contractible)*
- $A \times \mathsf{isProp}(A)$; *($A$ is a proposition with an element)*

Proof. Each of the above is a proposition: For $A \simeq 1$, this is a sum of propositions over a proposition; for $\mathsf{isContr}(A)$, we have already seen that this is a proposition in Theorem 2.6. Finally, we have $(A \times \mathsf{isProp}(A)) \to \mathsf{isProp}(A)$, and so then

$$(A \times \mathsf{isProp}(A)) \to \mathsf{isProp}(A \times \mathsf{isProp}(A)),$$

hence by Lemma 2.2, we have $A \times \mathsf{isProp}(A)$ is a proposition.

As the types are logically equivalent propositions (by Lemma 1.11), they are all equivalent. □

2.5 Resizing rules

One difference between MLTT and Topos logic is that Martin-Löf type theory is a predicative system which uses universe levels to avoid paradoxes of self-reference. However, the stratification of types into universe levels limits our ability to make certain arguments. For example, the type $A \to \mathsf{Prop}$, which we view as the power set of A, is parameterized by the universe level of Prop. In particular, we have for each universe level i, a type $\mathsf{Prop}_i : \mathcal{U}_{i+1}$, so that $A \to \mathsf{Prop}_i$ has type $\mathcal{U}_j$, where j is the maximum of $i+1$ and the universe level of A. On the other hand any topos has an object Ω of truth values, so that Ω^A is the power set of A. MLTT can be made impredicative by *resizing* propositions; we explain this below, and discuss it where relevant, but we do not make use of resizing principles anywhere in the main development.

In his Bergen lecture [90], Voevodsky introduced two resizing rules, which say informally,

1. if $P : \mathcal{U}_i$ is such that $\mathsf{isProp}(P)$, then in fact $P : \mathcal{U}_0$;
2. for any $\mathcal{U}_i$, we have that $\Sigma(A : \mathcal{U}_i), \mathsf{isProp}\, A : \mathcal{U}_0$.

These allow us to define $\Omega \stackrel{\mathsf{def}}{=} \mathsf{Prop}_0$, and ensure that no new propositions are added at higher levels. However, Voevodsky posited these as rules in the formal theory, and the consistency of these rules is an open question.

The HoTT book gives a weaker version of resizing. For any universe level i, we have an inclusion $\mathsf{Prop}_i \to \mathsf{Prop}_{i+1}$, where the type $\mathsf{Prop}_i \to \mathsf{Prop}_{i+1}$ is a type in $\mathcal{U}_{i+2}$. The HoTT book assumes that all of these inclusion maps are equivalences, so that all propositions can be replaced with a proposition in $\mathcal{U}_0$. This allows us to define $\Omega : \mathcal{U}_1$ by $\Omega \stackrel{\mathsf{def}}{=} \mathsf{Prop}_0$, and then we have an equivalence $\Omega \simeq \mathsf{Prop}_n$ in universe $\mathcal{U}_{n+1}$. Note that this weaker form of propositional resizing follows already from LEM, since LEM tells us that $\mathsf{Prop}_i \simeq 2$ for any universe level i.

If we have some type $\Omega : \mathcal{U}$ of propositions, then we may define the truncation for any $X : \mathcal{U}$ as $\|X\|$ as

$$\|X\| \equiv \Pi(P : \Omega), (X \to P) \to P.$$

We have $|-| : X \to \|X\|$ given by

$$|x| = \lambda P.\lambda f.f(x).$$

Now if P is any proposition, and we have $f : X \to P$, we can factor f through $|-|$ and a map $\|X\| \to P$ given by $w \mapsto f(P, w)$.

While we will sometimes discuss the consequences of resizing, we will not use it in any of the main development.

2.6 Homotopy levels and sets

Now that we've developed some of the theory of equivalences, we can direct our attention more fully to homotopy n-types, defined in Section 1.5. First, we note that being an n-type is in fact a proposition, and then we continue onto some closure properties.

Lemma 2.49. *For all X, we have $\mathsf{isProp}(\mathsf{isProp}(X))$ and moreover, for any n, we have $\mathsf{isProp}(\mathsf{is}\text{-}n\text{-}\mathsf{type}(X))$. That is, being a proposition or a homotopy n-type is a proposition.*

Proof. For isProp, this is Theorem 2.5 and function extensionality.

For the n-types, we proceed by induction: For $n = -2$, this is Lemma 2.2 and function extensionality.

For $n = k+1$, assume that $\mathsf{is}\text{-}k\text{-}\mathsf{type}$ is proposition-valued. Then $\Pi(x, y : X), \mathsf{is}\text{-}k\text{-}\mathsf{type}(x = y)$ is a proposition. □

Lemma 2.50. *If $e : A \to B$ is an embedding, and B is an $n+1$-type, then so is A for $n \geq -1$.*

Proof. We have that $(a = a') \simeq (e(a) = e(a'))$ and the later type is an n-type. □

Theorem 2.51. *If $r : A \to B$ is a retraction, and A is an n-type, then B is an n-type.*

Proof. For $n \geq -1$, this follows from the previous lemma, since sections are embeddings. For $n = -2$, this is Theorem 1.9. □

Theorem 2.52. *If $e : A \to B$ is an embedding, and B has decidable equality, then so does A.*

Proof. Let $a, a' : A$. Since e is an embedding, we have $(a = a') \simeq (e(a) = e(a'))$, and the latter has decidable equality. □

Lemma 2.53. *If $r : A \to B$ is a retraction and A has decidable equality, then so does B.*

Theorem 2.54. *The n-types are closed under equivalence: if $A \simeq B$ and B is an n-type, then so is A.*

Proof. As equivalences are in particular retractions, this follows from Theorem 2.51. □

Theorem 2.55. *The n-types are closed under Σ, for all n.*

Proof. Induction on n. For $n = -2$, it suffices to show that $\Sigma(a : 1), P(a)$ is contractible when each $P(a)$ is, and in turn it is enough to show that 1×1 is contractible, which is immediate. So let A be an $n+1$-type and let $B : A \to \mathcal{U}$ such that $B(a)$ is an $n+1$-type for all $a : A$. We have that

$$\big((a, b) = (a', b')\big) \simeq \Big(\Sigma(p : a = a'), b =^p_B b'\Big),$$

and this is an n-type by the inductive hypothesis. □

Theorem 2.56. *The n-types form an exponential ideal for each n. In particular if B is an n-type, then $A \to B$ is an n-type.*

Proof. By induction on n. For $n = -2$, this is function extensionality. Now let $B(a)$ be an $n+1$-type for each $a : A$, and let $f, g : \Pi(a : A), B(a)$. We have

$$(f = g) \simeq \big(\Pi(x : A), f(x) = g(x)\big),$$

and this is an n-type by the inductive hypothesis. □

In general, we will be interested in the level of sets and propositions. Importantly, this includes all types with decidable equality.

Lemma 2.57. *If A is a decidable type, then $(\neg\neg A) \to A$.*

Proof. By induction on the element $p : A + \neg A$. If $p = \mathsf{inl}\, a$, then $a : A$, and so $\lambda x.a : (\neg\neg A) \to A$. Otherwise, we have $p = \mathsf{inr}\, n$. But then if $q : \neg\neg A$, we have $q(n) : \emptyset$, and so we have

$$(\lambda q.!_A(q(n))) : (\neg\neg A) \to A.$$ □

Theorem 2.58 (Hedberg)**.** *If A has decidable equality, then A is a set.*

Proof. If A has decidable equality, then by Lemma 2.57 we have

$$\Pi(x, y : A), \neg\neg(x = y) \to (x = y).$$

As $\neg\neg X$ is a proposition for any X and $X \to \neg\neg X$ for any X, we have that $\neg\neg X$ is a retract of X whenever $\neg\neg X \to X$. In particular, if A has decidable equality, then for all $x, y : A$ the type

$\neg\neg(x = y)$ is a retract of $x = y$. Then by Lemma 2.22, we know that $\neg\neg(x = y) \simeq (x = y)$. Since $\neg\neg(x = y)$ is a proposition, we then have that A is a set. $\square$

N.B. the above proof is not Hedberg's original proof, and uses function extensionality, while the original proof doesn't.

Since we are particularly interested in sets, we collect the following facts, which we have already proved elsewhere.

Theorem 2.59. *The types $\emptyset$, 1, 2 and $\mathbb{N}$ are sets. Sets are closed under all basic type formers: If A is a set, and $B : A \to \mathcal{U}$ is such that $\Pi(a : A), \mathsf{isSet}(B(a))$, then $\Sigma(a : A), B(a)$ and $\Pi(a : A), B(a)$ are sets.*

Theorem 2.60. *If B is a set, then for any $f : A \to B$, the following are logically equivalent:*

1. *f is an embedding,*
2. *f is a monomorphism: For any $g, h : X \to A$ we have*

$$(f \circ g = f \circ h) \to (g = h).$$

Proof. Let f be an embedding and $g, h : X \to A$ such that $f \circ g = f \circ h$. Then for any $x : X$ we have $f(g(x)) = f(h(x))$, and since f is an embedding $g = h$.

Now let f be a monomorphism, and fix $x, y : A$ such that $f(x) = f(y)$. Define $c_x, c_y : 1 \to A$ as $c_x(u) \stackrel{\mathsf{def}}{=} x$ and $c_y(u) \stackrel{\mathsf{def}}{=} x$. As $f(x) = f(y)$, we have $f \circ c_x = f \circ c_y$, and so $c_x = c_y$, and $x = y$, so f is an embedding. $\square$

Corollary 2.61. *If B is a set, then for any $f : A \to B$, f being an embedding and f being a monomorphism are equivalent as types.*

Proof. As B is a sets, the type

$$\Pi(X : \mathcal{U}), \Pi(g, h : X \to A), (f \circ g = f \circ h) \to g = h,$$

is a proposition, so we have that the type witnessing that f is an embedding and the type witnessing that f is a monomorphism are logically equivalent propositions. $\square$

2.7 Higher-inductive types

There are two sorts of higher inductive types we will look at: *quotient inductive types*, which we will use for comparison of our developments with other approaches to partiality, and the *homotopy circle*, which we use as the prototypical example of a type which is not a set to show when assumptions on types are necessary. This appears, for example, in Section 5.11, when we discuss the difference between our notion of dominance and the usual one.

Quotient inductive types are given by specifying simultaneously an inductive type together with a quotient of that type by some relation. The motivating example is that of *set quotients*: given a type A and a relation $R : A \to A \to \mathsf{Prop}$, we define the smallest *set* A/R for which there is a map $A \to A/R$ respecting R. Explicitly, the type A/R is defined inductively by

- a *quotient map* $[-] : A \to A/R$;
- for each $x, y : A$ with $R(x, y)$, a path $[x] = [y]$.
- the *set truncation*: For each $x, y : A/R$, and $p, q : x = y$, a path $p = q$.

Then the recursion principle for A/R says that if B is any set, and we have a function $f : A \to B$ such that $R(x, y) \to f(x) = f(y)$, then there is a unique map $f/R : A/R \to B$ such that $f(a) = (f/R)([a])$ for all $a : A$.

Lemma 2.62. *For any type A and any relation $R : A \to A \to \mathsf{Prop}$, the quotient map $[-] : A \to A/R$ is surjective.*

Proof. For any element of the form $[a] : A/R$, we know have that $\mathsf{refl} : [a] = [a]$. For the higher constructors, we need to see that $|(a, \mathsf{refl})| = |(b, \mathsf{refl})|$ for any $a, b : A$ with $R(a, b)$, and that being a surjection is a set. Since being a surjection is a proposition, both of these facts are immediate. $\square$

The set truncation is necessary for this notion to be well-behaved: suppose we defined the quotient of A by R as above, but without the truncation constructor, and consider the quotient of 1 by the relation R with $R(\star, \star)$. This has as constructors,

- a point $b : 1/R$,
- a path $b = b$.

This type is exactly the homotopy circle S^1 given by Definition 2.64 below, which is not a set if univalence holds. Then we would have that the quotient of a set by a relation is no longer a set.

On the other hand, we can form the propositional truncation as the set-quotient by the total relation $\nabla \stackrel{\mathsf{def}}{=} \lambda x, y.1$. We can take the truncation to be X/∇ with the quotient map $X \to X/\nabla$ as the truncating map. Then any proposition P is in particular a set, and any map $X \to P$ must respect the equivalence relation, so it factors through X/∇.

It is more typical, however, to give the propositional truncation of X as a higher inductive type by directly expressing the universal property: the type $\|X\|$ is generated by

- a truncation constructor $|-| : X \to \|X\|$, and
- a family of path constructors $\Pi(x, y : \|X\|), (x = y)$.

There is a difference in the elimination principle for $\|X\|$ as a HIT and X/∇. We can only eliminate $\|X\|$ into propositions, while we can eliminate X/∇ into any set A, so long as we have a constant function $X \to A$. However, we will see in Section 3.3 that the elimination principle for X/∇ also holds for $\|X\|$.

The quotient by an equivalence relation can be defined concretely in a higher universe; with resizing, we can perform this truncation without raising universe levels. Define an *equivalence class of* R to be a predicate $P : A \to \mathsf{Prop}$ such that P is equivalent to $R(a, -)$ for an anonymous $a : A$:

$$\exists(a : A), \Pi(b : B), (P(b) \simeq R(a, b))\,.$$

Then define

$$A \mathbin{/\!\!/} R \stackrel{\mathsf{def}}{=} \Sigma(P : A \to \mathcal{U}), P \text{ is an equivalence class of } R.$$

Note that $A \mathbin{/\!\!/} R$ lives in $\mathcal{U}_1$. Nevertheless, we then have a map $q : A \to A \mathbin{/\!\!/} R$ given by

$$q(a) = \lambda b.R(a, b).$$

This map q respects R, and so there is a map $A/R \to A \mathbin{/\!\!/} R$.

Theorem 2.63. *For any type $A : \mathcal{U}$ and equivalence relation $R : A \to A \to \mathsf{Prop}$, the type $A /\!\!/ R$ is a set, the map $q : A \to A /\!\!/ R$ is a surjection which respects R, and moreover the induced map $s : A/R \to A /\!\!/ R$ is an equivalence.*

Proof. We have that $A \to \mathsf{Prop}$ is a set, and that being an equivalence class is a property, so $A /\!\!/ R$ is a set, as a subtype of a set.

Let $P : A /\!\!/ R$. We wish to see $\exists(a : A), P = q(a)$. As $q(a) = R(a, -)$, this is the definition of being an equivalence class.

Next we need to see that if $R(a, a')$, then $q(a) = q(a')$. For a fixed b, we have $q(a)(b) = R(a, b)$ and $q(a')(b) = R(a', b)$, so we need to see that $R(a, b) = R(a', b)$. As both are propositions, we need $R(a, b) \Leftrightarrow R(a', b)$. We have maps in both directions by transitivity and symmetry of R.

Finally, we need to see that the induced map $s : A/R \to A /\!\!/ R$ is an equivalence. First, s is surjective: as q is surjective, for any equivalence relation P, we have

$$\exists(a : A), q(a) = P,$$

and q factors as $s \circ [-]$, so we have

$$\exists(a : A), s[a] = P.$$

Finally, s is an embedding: let $x, y : A/R$ and let $s(x) = s(y)$. As $x = y$ is a proposition, and $[-]$ is surjective, we may assume that $x = [a]$ and $y = [b]$ for some $a, b : A$. Then we have for any c that

$$R(a, c) = s([a])(c) = s(x)(c) = s(y)(c) = s([b])(c) = R(b, c),$$

so that $R(a, b) = R(b, b)$, and the latter is inhabited by reflexivity. □

We turn now to the homotopy circle, which we use to show deficiencies in proposed notions. The basic idea is that an obvious way of stating a particular notion may not interact as expected with general types, and we use S^1 to show that we need to be more subtle when defining a particular notion. This appears most clearly in Section 5.11, where we see why we cannot translate definitions from synthetic domain theory directly.

Definition 2.64. The *homotopy circle*, $\mathsf{S}^1 : \mathcal{U}$, is inductively generated by

- an element $\mathsf{base} : \mathsf{S}^1$,
- a path $\mathsf{loop} : \mathsf{base} = \mathsf{base}$.

The recursion principle is easy to state:

- For any type $C : \mathcal{U}$, with $b : C$ and $l : b = b$, we have a unique map $f : \mathsf{S}^1 \to C$ such that

$$f(\mathsf{base}) \equiv b$$

and

$$\mathsf{ap}_f(\mathsf{loop}) = l.$$

With ordinary inductive types, it is safe to treat the computation rules associated with recursion and induction principles as judgemental equalities. For higher inductive types, there are philosophical and technical issues with treating the computation rules as judgemental equalities. More information on the status of higher-inductive types can be found in the discussion at the end of the chapter (Section 2.8).

The induction principle has the same shape, but is unexpectedly trickier to state explicitly. The key point is that the image of loop is no longer a simple path, but a path living over loop:

- For any $C : \mathsf{S}^1 \to \mathcal{U}$ with $b : C(\mathsf{base})$ and $l : \mathsf{transport}^C(\mathsf{loop}, b) = b$, we have a unique $f : \Pi(x : \mathsf{S}^1), C(x)$ such that

$$f(\mathsf{base}) \equiv b$$

and

$$\mathsf{apd}_f(\mathsf{loop}) = l.$$

Note that here $\mathsf{apd}_f(\mathsf{loop})$ must have type $\mathsf{transport}^C(\mathsf{loop}, b) = b$, so the required equation for the image of loop does type-check. The induction principle can be stated another way, which we give as a lemma.

Lemma 2.65. *Given any* $C : \mathsf{S}^1 \to \mathcal{U}$*, we have that*

$$\big(\Pi(x : \mathsf{S}^1), C(x)\big) \simeq \Big(\Sigma(b : C(\mathsf{base})), \mathsf{transport}^B(\mathsf{loop}, b) = b\Big).$$

That is, dependent functions from S^1 to a type family C are exactly given by an element of $C(\mathsf{base})$ and a path over loop in C.

Proof. Given $f : \Pi(x : \mathsf{S}^1), C(x)$ define $b \overset{\mathsf{def}}{=} f(\mathsf{base})$ and $p \overset{\mathsf{def}}{=} \mathsf{apd}_f(\mathsf{loop})$, so that we have the pair $(b, p) : \Sigma(b : C(\mathsf{base})), \mathsf{transport}^C(\mathsf{loop}, b) = b$. This gives us a map

$$c : \big(\Pi(x : \mathsf{S}^1), C(x)\big) \to \Sigma(b : C(\mathsf{base})), \mathsf{transport}^C(\mathsf{loop}, b) = b.$$

Now if (b, p) is any element of $\Sigma(b : C(\mathsf{base})), \mathsf{transport}^C(\mathsf{loop}, b) = b$, we have that the dependent function $f : \Pi(x : \mathsf{S}^1), C(x)$ defined by $f(\mathsf{base}) = b$ and $\mathsf{apd}_f(\mathsf{loop}) = p$ is the unique function f such that $c(f) = (b, p)$. Hence, c has contractible fibers. □

The utility of S^1 as a counterexample stems from the fact that it is not a set:

Lemma 2.66. *Assuming univalence,* S^1 *is not a set.*

There are several ways to prove this; in the HoTT Book, they show that S^1 is equivalent to a type (in a higher universe) which is not a set. We instead follow the proof that $\pi_1(\mathsf{S}^1) = \mathbb{Z}$. The type-theoretical proof is an encode-decode proof (See Section 1.7).

Proof. We show that $\mathsf{base} = \mathsf{base}$ is equivalent to $\mathbb{Z}$, by giving a type family

$$\mathsf{code} : \mathsf{S}^1 \to \mathcal{U},$$

along with a family of functions

$$\mathsf{encode} : \Pi(x : \mathsf{S}^1), (\mathsf{base} = x) \to \mathsf{code}(x)$$

and a family of inverses

$$\mathsf{decode} : \Pi(x : \mathsf{S}^1), \mathsf{code}(x) \to (\mathsf{base} = x).$$

We define

$$\begin{aligned}\mathsf{code}(\mathsf{base}) &\stackrel{\mathsf{def}}{=} \mathbb{Z},\\ \mathsf{apd}_{\mathsf{code}}(\mathsf{loop}) &= \mathsf{ua}(\mathsf{succ}).\end{aligned}$$

So that if $\mathsf{code}(\mathsf{base}) \simeq (\mathsf{base} = \mathsf{base})$ we have that $\mathsf{base} = \mathsf{base}$ is equivalent to a set which is not a proposition. Observe that we have $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, z) = z + 1$ as follows:

$$\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, z) = \mathsf{idtofun}(\mathsf{apd}_{\mathsf{code}}(\mathsf{loop}), z) = \mathsf{idtofun}(\mathsf{ua}(\mathsf{succ}), z) = z + 1.$$

Similarly, we have $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1}, z) = z - 1$.

We define encode directly, and decode by induction:

$$\begin{aligned}\mathsf{encode}(p) &\stackrel{\mathsf{def}}{=} \mathsf{transport}^{\mathsf{code}}(p, 0)\\ \mathsf{decode}_{\mathsf{base}}(z) &= \mathsf{loop}^{z}.\end{aligned}$$

For the loop case of decode, we need an element of

$$\mathsf{transport}^{\lambda x.\mathsf{code}(x)\to(\mathsf{base}=x)}(\mathsf{loop}, \mathsf{loop}^{-}) = \mathsf{loop}^{-}.$$

We have

$$\begin{aligned}\mathsf{transport}^{\lambda x.\mathsf{code}(x)\to(\mathsf{base}=x)}(\mathsf{loop}, \mathsf{loop}^{-}) &= \mathsf{transport}^{\lambda x.\to(\mathsf{base}=x)}(\mathsf{loop}) \circ \mathsf{loop}^{-} \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1})\\ &= (\lambda p.p \cdot \mathsf{loop}) \circ \mathsf{loop}^{-} \circ \mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^{-1})\\ &= (\lambda p.p \cdot \mathsf{loop}) \circ \mathsf{loop}^{-} \circ \mathsf{pred}\\ &= \lambda n.\mathsf{loop}^{n-1} \cdot \mathsf{loop}\\ &= \lambda n.\mathsf{loop}^{n}.\end{aligned}$$

We need to see that encode and decode are inverse. To show that $\mathsf{decode}_x(\mathsf{encode}_x(p)) = p$ for all $x : \mathsf{S}^1$ and $p : \mathsf{base} = x$ we use based path induction:

$$\mathsf{decode}_{\mathsf{base}}(\mathsf{encode}_{\mathsf{base}}(\mathsf{refl})) = \mathsf{decode}(\mathsf{transport}^{\mathsf{code}}(\mathsf{refl}, 0)) = \mathsf{decode}(0) = \mathsf{loop}^{0} = \mathsf{refl}.$$

For the other direction, we use the induction principle for S^1. The claim is that for all $z : \mathbb{Z}$ we have $\mathsf{encode}_{\mathsf{base}}(\mathsf{decode}_{\mathsf{base}}(z)) = z$. That is, we want to see $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^z, 0) = z$, but we remarked earlier that $\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}, x) = \mathsf{succ}(x)$, and so we have

$$\mathsf{transport}^{\mathsf{code}}(\mathsf{loop}^z, 0) = 0 + z = z. \qquad \square$$

Note that even without univalence, we cannot have that S^1 is a set, since our type theory is still consistent with univalence. This means we can still use S^1 to see whether a notion behaves well with general types.

Lemma 2.67. *We have $\Pi(x, y : \mathsf{S}^1), \|x = y\|$.*

Proof. By induction on both x and y: fix $x \equiv y \equiv \mathsf{base}$. Then we have $\mathsf{refl} : x = y$. Truncating this element gives $|\mathsf{refl}| : \|x = y\|$. We need to give a path over loop in both coordinates from $|\mathsf{refl}|$ to $|\mathsf{refl}|$, but since $\|x = y\|$ is a proposition, there is nothing to check. $\square$

Corollary 2.68. *If Y is a set and $f : \mathsf{S}^1 \to Y$, then for any $x, y : \mathsf{S}^1$ we have that $f(x) = f(y)$.*

Proof. Fix $x, y : \mathsf{S}^1$. By the previous lemma, we have $\|x = y\|$. As Y is a set, $f(x) = f(y)$ is a proposition, so we may assume we have $p : x = y$. Then $\mathsf{ap}_f(p) : f(x) = f(y)$. $\square$

2.8 Discussion

Truncation was already being studied [60, 65] as *squash types* and *bracket types* before the univalent notion of proposition was introduced. In fact, Awodey and Bauer [4] proposed propositions as bracket types already in 2004, in the context of extensional type theory. When the general notion of higher-inductive type was introduced, they quickly became a standard example. Higher inductive types are, however, poorly understood. The schema is intuitively clear, but a mathematically satisfying account of their syntax and semantics is surprisingly difficult. Lumsdaine and Shulman recently released an awaited approach to the semantics of HITs via *cell-monads with parameters* [54], but these only work for HITs of a certain form, and it is not clear if there is a syntactic scheme that corresponds precisely to the semantic notions they give. A semantic view of HITs from the cubical approach is given in [20], and a view of HITs, focusing on those that exist at the level of groupoids (which is enough for the HITs we are interested in here) is in [27]. On the syntactic side, recent progress can be seen in [45] and in the more sweeping [15], which also contains a more thorough summary of the state of the art.

Higher-inductive types represent one standard approach to truncation, while propositional resize represents another. It is open whether Voevodsky's resizing rules are consistent with MLTT [23]. The weaker form of resizing discussed in the HoTT Book holds in the simplicial set model, and in fact in any model which validates the law of excluded middle.

The univalence axiom was first proposed by Vladimir Voevodsky, motivated by his simplicial set model [46]. However, it was noticed (under the name *universe extensionality*) in the groupoid model already by Hofmann and Streicher [37] in 1995, although all types in the universe are sets in the groupoid model. The notion of h-level is also due to Voevodsky. The first proof of function extensionality from univalence was given by Voevodsky in his Foundations library [91], and it has been been modified and reworked in various ways since. The proof of function extensionality from naive function extensionality was first done by Voevdosky in the Foundations library, but seems to have been overlooked until Martín Escardó found it and translated it to Agda in his TypeTopology library [29].

Extensionality axioms, such as univalence, have always been treated with some trepidation in type theory, since axioms break computational properties like canonicity—the metatheorem that all terms of type $\mathbb{N}$ are judgementally equal to a numeral. Since the outset, there has been hope that the univalence axiom can be given a computational interpretation. Following the introduction of the first model in cubical sets [40, 8], a great deal of work has been done on developing *cubical type theories* [18] which take paths as primitive, and in which the univalence axiom is a theorem. Importantly, canonicity holds in cubical type theory [39].

CHAPTER 3

Mathematical foundations

In the previous chapter, we developed the core ideas of univalent mathematics. Here we turn to the more traditional mathematical notions we will need in part 2. Our approach is still heavily informed by the univalent perspective. In particular, choice principles in Section 3.6 are stated in a form which is unique to univalent mathematics. We begin by examining relations and a predicative notion of power set (Section 3.2). Importantly, we will see (in Theorem 3.6) that assuming the univalence axiom, functions are single-valued relations, which will be a prototype of similar results about partial maps in Chapter 5. We then discuss a notion of constancy introduced in [47] that will appear in a few places, before turning to the natural numbers. The natural numbers satisfy a property that looks superficially like Markov's principle, which allows us to remove truncations in certain cases. This is an interesting example, and will be crucial when working with computability in Part III; we also introduce here a representation of a predicate on $\mathbb{N}$ as a function $\mathbb{N} \to \mathbb{N}$ (which we give the ad hoc name *characteristic function*) that will be used to define computability of predicates. Before turning to choice, we discuss an internal notion of monad via Kleisli extension.

3.1 Assumptions

In the previous chapter, we introduced several extensions of MLTT which are used in univalent approaches to mathematics. Here we lay out explicitly the assumptions we use in the rest of the thesis.

Each of the following assumptions is parameterized by a universe $\mathcal{U}$, and we assume them for all universes.

Assumption 1 (Function extensionality)**.** Propositions form a strong exponential ideal in every universe.

Assumption 2 (Proposition extensionality)**.** For every universe $\mathcal{U}$ and for every propositions $P, Q : \mathcal{U}$, if P and Q are logically equivalent, then $P = Q$.

Assumption 3 (Propositional truncation)**.** For every universe $\mathcal{U}$ and any type $X : \mathcal{U}$ we have a proposition $\|X\| : \mathcal{U}$ and a map $|-| : X \to \|X\|$ universal among all propositions which X maps into.

When we need univalence, we will assume it explicitly. We will not assume resizing rules, but we occasionally discuss their consequences. Similarly, we will not use any higher inductive types besides truncations, but we will sometimes discuss their properties. In particular, we will compare our work to work by others using truncations (see Sections 4.1 and 4.2, as well as Section 5.7), and we will look at the circle for motivation in Section 5.11.

3.2 Predicates and relations

In category theory, a *subobject* of an object X is an equivalence class of monos into X. Replacing mono with embedding (Section 2.3), we could take a subobject to be an equivalence class of embeddings into X. Since we should write the type of embeddings into X as

$$\Sigma(A : \mathcal{U}), \Sigma(f : A \to X), \mathsf{isEmbedding}(f),$$

an equality between embeddings $(A, f, -)$ and $(B, g, -)$ would consist of a path $p : A = B$ such that $g \circ \mathsf{idtofun}(p) = f$. If we have univalence, equality here is exactly the equivalence relation we need; if not we should replace equality in the first component with equivalence as types.

With univalence, the equivalence relation of interest is simply equality, so there is no quotienting to be done. However, even without univalence, for any embedding $f : A \to X$ we can find a canonical representative of its equivalence class, by looking at the total space of the fibers over f. In fact, this suggests a definition of subtype which works better in our framework: predicates on X. Explicitly, given any embedding $f : A \to X$, we have the predicate $\mathsf{fib}_f : X \to \mathcal{U}$, which is proposition valued precisely because f is an embedding. Conversely, given any predicate $P : X \to \mathcal{U}$ the first projection $\big(\Sigma(x : X), P(x)\big) \to X$ is an embedding.

Lemma 3.1. *For any $X : \mathcal{U}$, the type of predicates $X \to \mathsf{Prop}$ is a retract of the type*

$$\Sigma(A : \mathcal{U}), \Sigma(f : A \to X), \mathsf{isEmbedding}(f)$$

of embeddings into X. Moreover, assuming univalence, the retraction is an equivalence.

Proof. Let $f : A \to X$ be an embedding and define $R : X \to \mathcal{U}$ by

$$R(x) \stackrel{\mathsf{def}}{=} \mathsf{fib}_f(x).$$

As f is an embedding, R is proposition valued. Conversely, if R is a predicate, define

$$A \stackrel{\mathsf{def}}{=} \Sigma(x : X), R(x),$$

with $f : A \to X$ the first projection. The fiber of the first projection at x is

$$\Sigma((x', r') : \Sigma(x' : X), R(x)), x' = x,$$

which is equivalent to $R(x)$ by Lemma 1.24. Note that by proposition extensionality and function extensionality, we have that $\mathsf{fib}_{\mathsf{pr}_0} = R$, so that $(A, f) \mapsto \mathsf{fib}_f(x)$ is a retraction. Moreover, by Lemma 2.10, we know that $A \simeq \Sigma(x : X), \mathsf{fib}_f(x)$ and that this equivalence composes with the projection to give f, so that by univalence, $(A, f) = (\Sigma(x : X), \mathsf{fib}_f(x), \mathsf{pr}_0)$. $\square$

With this in mind, we make the following definition.

Definition 3.2. A *subtype* of a type X is a map $X \to \mathsf{Prop}$. If X is a set, we will use the term *subset* interchangeably with *subtype*. We will call the type of subtypes of X the *power set* of X and write

$$\mathcal{P}X \stackrel{\mathsf{def}}{=} X \to \mathsf{Prop}\,.$$

If $A : \mathcal{P}X$ we sometimes write $x \in A$ for the predicate $A(x)$. Following our terminology that a type Y is inhabited when $\|Y\|$, we say that A *is an inhabited* subtype of X when

$$\|\Sigma(x : X), x \in A\|\,.$$

The term power *set* is indeed justified:

Lemma 3.3. *The power set of X is always a set.*

Proof. We know that Prop is a set, so by Lemma 2.56, $X \to \mathsf{Prop}$ is a set. $\square$

In a traditional set-theoretic formalization, relations are given either as formulas, or as subsets of a cartesian product, and then functions $f : X \to Y$ are defined as relations for which there is a unique $y : Y$ such that $f(x, y)$. While functions are primitive in type theory, we can show that this correspondence holds, but to account for general types, we must replace relations with arbitrary type families.

Definition 3.4. A type family $R : X \to Y \to \mathcal{U}$ is *functional* when for each $x : X$, there is a unique $y : Y$ such that $R(x, y)$. That is, when

$$\Pi(x : X), \mathsf{isContr}(\Sigma(y : Y), R(x, y)).$$

This definition demonstrates a peculiar feature of univalent definitions. The traditional way to define a *functional relation* would be to say that R is functional when for each $x : X$ there is $y : Y$ such that $R(x, y)$ and whenever $R(x, y)$ and $R(x, y')$ then $y = y'$. This becomes,

$$\Pi(x : X), \Sigma(y : Y), R(x, y) \times \Pi(y, y' : Y), R(x, y) \to R(x, y') \to y = y'.$$

This definition is inadequate in a univalent framework. In the case where Y is not a set, and in the case where R is not valued in propositions, the type above may have many elements, but we would like to use being functional as a property: we would like not only that there is a unique $y : Y$ such that $R(x, y)$, but that there is a unique pair $(y, r) : \Sigma(y : Y), R(x, y)$. This situation is captured by contractibility.

In the case where Y is a set, any functional type family is already a relation, so the two types above are equivalent.

Lemma 3.5. *If Y is a set, and $A : Y \to \mathcal{U}$ such that $\mathsf{isProp}(\Sigma(y : Y), A(y))$, then Y is valued in propositions. That is,*

$$\Pi(y; Y), \mathsf{isProp}(A(y)).$$

Proof. Fix y so that we have

$$A(y) \simeq \Sigma((y', a) : \Sigma(y' : Y), A(y')), y = y',$$

by Lemma 1.24, and this is the sum of propositions indexed by a proposition. $\square$

Given a type family $R : X \to Y \to \mathcal{U}$ with a witness w that R is functional, there is a function $f_R : X \to Y$ defined by

$$f_R(x) \stackrel{\mathsf{def}}{=} \pi_0(w(x)).$$

Conversely, given any $f : X \to Y$ there is a type family $R_f : X \to Y \to \mathcal{U}$ given by

$$R_f(x, y) \stackrel{\mathsf{def}}{=} f(x) = y.$$

Then $\Sigma(y : Y), f(x) = y$ is the singleton based at $f(x)$, so is contractible. In the presence of univalence, this gives an equivalence between the type of functional relations and the type of functions. And using Lemma 3.5, if Y is a set, then univalence is not needed.

Theorem 3.6 (Rijke, [73]). *For any $X, Y : \mathcal{U}$, the map taking a functional $R : X \to Y \to \mathcal{U}$ to f_R is a left-inverse of the map taking $f : X \to Y$ to R_f. Moreover, for any $x : X$ and $y : Y$, the type $R_{f_R}(x, y)$ is equivalent to $R(x, y)$.*

Hence, assuming univalence, the map $f \mapsto R_f$ gives rise to an equivalence

$$(X \to Y) \simeq \Sigma(R : X \to Y \to \mathcal{U}), \Pi(x : X), \mathsf{isContr}(\Sigma(y : Y), R(x, y)).$$

Moreover, if Y is a set, univalence is not needed in the proof.

Proof. For any $f : X \to Y$ we have $f_{R_f}(x) = f(x)$ by definition, since $(f(x), \mathsf{refl})$ is the center of contraction of $\Sigma(y : Y), f(x) = y$.

For the other direction, we wish to see that for any x and y,

$$R_{f_R}(x, y) \simeq R(x, y).$$

By Theorem 1.28, it is enough to find an equivalence

$$\big(\Sigma(x : X), R(x, y)\big) \simeq \big(\Sigma(x : X), R_{f_R}(x, y)\big),$$

which is the identity on the first component.

Now, suppose Y is a set. Let us first verify that not only is $\Sigma(y : Y), R(x, y)$ a proposition, but also that $R(x, y)$ is a proposition for all $y : Y$. With x fixed, define $A : Y \to \mathcal{U}$ to be $A(y) \overset{\mathsf{def}}{=} R(x, y)$, so that by Lemma 3.5 we know $\mathsf{isProp}(A(y))$ for all $y : Y$. That is, $\mathsf{isProp}(R(x, y))$ for all $y : Y$ as desired.

Letting x vary again, this means that $\Pi(x : X), \Pi(y : Y), \mathsf{isProp}(R(x, y))$. Similarly, since Y is a set, for any $f : X \to Y$, and elements $x : X$ and $y : Y$ the type $f(x) = y$ is a proposition, and so both $R_{f_R}(x, y)$ and $R(x, y)$ are propositions. Then the result follows from Theorem 3.6 by proposition extensionality, since $R_{f_R}(x, y) \simeq R(x, y)$. $\square$

3.3 Constancy

This section covers material from [47, 49]. Our goal is to show that $\|X\|$ has the universal property of the quotient of X by the total relation $\lambda x, y.1 : X \to X \to \mathcal{U}$. That is, we can map out of $\|X\|$ to sets, even though the universal property of truncation only tells us how to define maps to propositions.

Definition 3.7. A function $f : A \to B$ is *constant* if for all $x, y : A$, we have $f(x) = f(y)$.

Constant functions are called *weakly constant* in [47, 49].

Theorem 3.8. *Let $X, Y : \mathcal{U}$ and let $P : \mathcal{U}$ be a type for which $P \to Y$. If X implies that P is contractible, then $\|X\| \to Y$. In particular, if $f : X \to Y$ is a function which factors through P, then f factors through $\|X\|$.*

Proof. Consider the following diagram.

$$\begin{tikzcd}
X \arrow[r] \arrow[d, "|-|"'] & \mathsf{isContr}(P) \arrow[d, "\mathsf{pr}_0"] & \\
\|X\| \arrow[ur, dashed] & P \arrow[r] & Y
\end{tikzcd}$$

As $\mathsf{isContr}(P)$ is a proposition, we have the dashed function $\|X\| \to \mathsf{isContr}(P)$; composition gives $\|X\| \to Y$.

Now let $f : X \to Y$ factor through P as $g \circ h$. As $\|X\| \to \mathsf{isContr}(P)$, we have that $\|X\| \times P$ is a proposition, and moreover we have $X \to \|X\| \times P$, given by $x \mapsto (|x|, h(x))$. Since $\|X\| \times P$ is a proposition, we have a factorization of f as

$$\begin{tikzcd} X \arrow[r, "|-|"] \arrow[rrr, bend right, "h"'] & \|X\| \arrow[r] & \|X\| \times P \arrow[r, "\mathsf{pr}_1"] & P \arrow[r, "g"] & Y. \end{tikzcd}$$ □

Note that if P is a proposition and a function $f : X \to Y$ factors through P, then X implies that P is contractible. Then if $f : X \to Y$ factors through any proposition P, then f factors through $\|X\|$.

Theorem 3.9. *If $f : X \to Y$ is constant and Y is a set, then $\|X\| \to Y$.*

Proof. We know that f factors through $\mathsf{im}(f)$, so it suffices to show that when Y is a set and f is constant, then $\mathsf{im}(f)$ is a proposition.

Let $(y, u), (y', u') : \Sigma(y : Y), \|\Sigma(x : X), f(x) = y\|$. Since the second component inhabits a proposition, we need only a path $y = y'$. As y is a set, we know that $y = y'$ is a proposition, so we may untruncate our assumptions $u : \|\mathsf{fib}_f(y)\|$ and $u' : \|\mathsf{fib}_f(y')\|$ and use explicit witnesses $(x, p) : \Sigma(x : X), f(x) = y$ and $(x', p') : \Sigma(x : X), f(x) = y'$. Then we have

$$y \overset{p}{=} f(x) = f(x') \overset{p'}{=} y',$$

where the middle equality comes from constancy of f.

Explicitly, the above argument above gives

$$\Pi(y, y' : Y), \mathsf{fib}_f(y) \to \mathsf{fib}_f(y') \to y = y',$$

and since $y = y'$ is a proposition, we conclude

$$\Pi(y, y' : Y), \|\mathsf{fib}_f(y)\| \to \left\|\mathsf{fib}_f(y')\right\| \to y = y'.$$

By reshuffling and uncurrying, we get

$$\Pi(u, v : \mathsf{im}(f)), u = v.$$

Then f factors through a proposition, so by Theorem 3.8, we are done. □

The factorization of constant functions through propositions is related to another fact about constancy: if f is constant, then so is ap_f.

Lemma 3.10. *If $f : X \to Y$ is constant, then $\mathsf{ap}_f : x = x' \to f(x) = f(x')$ is constant for all $x, x' : X$. In particular, $\mathsf{ap}_f(p) = \mathsf{refl}_{f(x)}$ whenever $p : x = x$.*

Proof. Let k be a witness that f is constant. By path induction, for any $p : x = x'$ we have that $\mathsf{ap}_f(p) = k_{x,x}^{-1} \cdot k_{x,x'}$. Indeed, when p is reflexivity, we have

$$\mathsf{refl}_{f(x)} = k_{x,x}^{-1} \cdot k_{x,x}.$$

Now if $p, q : x = x'$, then

$$p = (k_{x,x}^{-1} \cdot k_{x,x'}) = q.$$

□

3.4 Predicates on $\mathbb{N}$

Since we will be focused on computability theory, we now turn to predicates on $\mathbb{N}$.

Lemma 3.11. *The standard ordering on $\mathbb{N}$ is decidable and is a total order.*

Lemma 3.12 (Bounded search)**.** *Let P be a decidable predicate on $\mathbb{N}$. Then for all $n : \mathbb{N}$, the types*

$$\Sigma(k : \mathbb{N}), P(k) \times (k \leq n)$$

and

$$\Sigma(k : \mathbb{N}), P(k) \times (k < n)$$

are decidable.

Proof. The proof is straightforward by induction: for 0 it holds immediately, since

$$(\Sigma(k : \mathbb{N}), P(k) \times (k \leq 0)) \simeq P(0).$$

For $n+1$, we have

$$(\Sigma(k:\mathbb{N}), P(k) \times (k \leq n+1)) \simeq P(n+1) + (\Sigma(k:\mathbb{N}), P(k) \times (k \leq n))$$

If $P(n+1)$ holds, we're done. Otherwise, check $\Sigma(k:\mathbb{N}), P(k) \times (k \leq n)$, which is decidable by the inductive hypothesis. $\square$

Lemma 3.13. *If P is a decidable predicate on $\mathbb{N}$, then*

$$(\exists(n:\mathbb{N}), P(n)) \simeq \Sigma(n:\mathbb{N}), (P(n) \times \Pi(k:\mathbb{N}), P(k) \to n \leq k)$$

In particular, we have $\|\sum_{n:\mathbb{N}} P(n)\| \to \sum_{n:\mathbb{N}} P(n)$.

Proof. Let $Q \overset{\text{def}}{=} \Sigma(n:\mathbb{N}), (P(n) \times \Pi(k:\mathbb{N}), P(k) \to n \leq k)$.

We proceed in 3 steps: first we show $(\Sigma(n:\mathbb{N}), P(n)) \to Q$ using bounded search. Next, we show that Q is a proposition, so that we can apply the universal property of truncation to get a map

$$\|\Sigma(n:\mathbb{N}), P(n)\| \to Q.$$

Finally, we project out of Q to see that $Q \to \Sigma(n:\mathbb{N}), P(n)$.

First, observe that under our assumptions, we have that for all k,

$$(\neg\Sigma(j:\mathbb{N}), P(j) \times j < k) \simeq (\Pi(j:\mathbb{N}), P(j) \to k \leq j)\,.$$

By bounded search, the predicate

$$P'(n) \overset{\text{def}}{=} \Sigma(k:\mathbb{N}), P(k) \times (k \leq n) \times (\neg\Sigma(j:\mathbb{N}), P(j) \times j < k)$$

is decidable. Let n be such that $P(n)$. If $P'(n)$ fails, then a quick argument shows that $P(n)$ must also fail, so we must have $P'(n)$. Let k be such that $P(k) \times (k \leq n) \times (\neg\Sigma(j:\mathbb{N}), P(j) \times j < k)$. Then we also have $m : \Pi(j:\mathbb{N}), P(j) \to k \leq j$, and $(k,(p,m)) : Q$, where $p : P(k)$.

If $(k, w) : Q$ and $(k', w') : Q$, we must have $k \leq k' \leq k$, so we have $p : k = k'$. Moreover, $P(k) \times \Pi(j : \mathbb{N}), P(j) \to k \leq j$ is a proposition, so we get $\mathsf{transport}(p, w) = w'$. That is, Q is a proposition, and $\Sigma(n : \mathbb{N}), P(n)$ implies Q, so then does $\exists(n : \mathbb{N}), P(n)$.

Finally, we have the map $Q \to \Sigma(n : \mathbb{N}), P(n)$ defined by

$$(k, (p, m)) \mapsto (k, p). \qquad \square$$

It is tempting to think of propositions as types which contain no more information than their inhabitedness. The above shows that this is misleading, as the type

$$\Sigma(n : \mathbb{N}), P(n) \times \Pi(k : \mathbb{N}), P(k) \to n \leq k$$

contains useful computational content. Moreover, we see here and in the definition of equivalence that there are predicates for which $(\exists(x : A), P(x)) \to \Sigma(x : A), P(x)$. In other words, if there exists an x such that $P(x)$, then we can find one explicitly. The proof of Lemma 3.13 can be found in [38], where they explicitly assume function extensionality in the proof. However, they sketch another way to untruncate $\|\Sigma(n : \mathbb{N}), P(n)\|$, without using function extensionality. The other argument uses ideas from Section 3.3. For an arbitrary type X, if there is a constant function $f : X \to X$, then the type of fixed points of f has the universal property of the truncation.

For any $f : X \to X$, define the type

$$\mathsf{fix}(f) \stackrel{\mathsf{def}}{=} \Sigma(x : X), x = f(x).$$

Lemma 3.14 (Fixed Point Lemma[49]). *Let X be a type and $f : X \to X$ be constant. Then* $\mathsf{fix}(f) \simeq \|X\|$.

Proof. Fix $k : \Pi(x, y : X), f(x) = f(y)$.

First, we need to see that $\mathsf{fix}(f)$ is a proposition. Given $(x, p), (y, q) : \mathsf{fix}(f)$ we have the path $r : x = y$ given by

$$x \stackrel{p}{=} f(x) \stackrel{k_{x,y}}{=} f(y) \stackrel{q^{-1}}{=} y.$$

We want a path $t : x = y$ such that $\mathsf{transport}(t, p) = q$, and r does not suffice, as we can compute $\mathsf{transport}(r, p) = r^{-1} \cdot p$ (in a moment, we give this computation for our path of interest). By Theorem 1.20, for any $t : x = y$ and $p : x = f(x)$, $\mathsf{transport}^{\lambda x.x=f(x)}(t, p) = t^{-1} \cdot p \cdot \mathsf{ap}_f(t)$. So defining $t \overset{\mathsf{def}}{=} p \cdot \mathsf{ap}_f(r) \cdot q^{-1}$, we have

$$\mathsf{transport}(t, p) = (q \cdot \mathsf{ap}_f(r^{-1}) \cdot p^{-1}) \cdot p \cdot \mathsf{ap}_f(p \cdot \mathsf{ap}_f(r) \cdot q^{-1}).$$

By Lemma 3.10, we can say that $\mathsf{ap}_f(p \cdot \mathsf{ap}_f(r) \cdot q^{-1}) = \mathsf{ap}_f(r)$, and so we have

$$q \cdot \mathsf{ap}_f(r^{-1}) \cdot p^{-1} \cdot p \cdot \mathsf{ap}_f(r) = q \cdot \mathsf{ap}_f(r^{-1}) \cdot \mathsf{ap}_f(r) = q.$$

Then, we have $(x, p) = (y, q)$ with witness given by our path t and the above argument. That is, $\mathsf{fix}(f)$ is a proposition.

We have the projection map $\mathsf{fix}(f) \to X$, as well as a map $X \to \mathsf{fix}(f)$ given by

$$x \mapsto (f(x), k(f(x), x)).$$

That is, $\mathsf{fix}(f)$ is a proposition which is logically equivalent to X. The theorem follows by Lemma 2.13. $\square$

Now fix a decidable predicate $P : \mathbb{N} \to \mathcal{U}$. This factors through a map $f : \mathbb{N} \to \mathbb{N}$, which takes value 0 at x if $\neg R(x)$ and takes value 1 at x if $P(x)$. Consider the type

$$X \overset{\mathsf{def}}{=} \Sigma(n : \mathbb{N}), f(n) = 1.$$

We can define a map $f : X \to X$ as follows: for $(n, w) : \Sigma(n : \mathbb{N}), f(n) = 1$, we find the least $k \leq n$ with $v : f(k) = 1$, using a bounded search. Take $f(n, w) = (k, v)$. Since there is at most one least k such that $f(k) = 1$, this function f is constant. Then by Lemma 3.14, we have that

$$\mathsf{fix}(f) \simeq \|\Sigma(n : \mathbb{N}), f(n) = 1\|.$$

Because $\mathsf{fix}(f) \Leftrightarrow X$, we then get a map $\|\Sigma(n : \mathbb{N}), f(n) = 1\| \to \Sigma(n : \mathbb{N}), f(n) = 1$.

In the above, we used that a decidable predicate $P : \mathbb{N} \to \mathcal{U}$ factors as $2 \circ p$ where 2 is the inclusion $2 \to \mathsf{Prop}$, and some function $p : \mathbb{N} \to 2$. The standard inclusion $i : 2 \to \mathbb{N}$ is a section of the function $\mathsf{pos} : \mathbb{N} \to 2$ taking 0 to 0 and all positive numbers to 1. Then P factors through a map $\chi_P : \mathbb{N} \to \mathbb{N}$ which we will call a *characteristic function* for P. By function extensionality, there is a unique characteristic function for P. We can put this together as follows

Definition 3.15. A *characteristic function* for a predicate $P : \mathbb{N} \to \mathsf{Prop}$ is a function $\chi_P : \mathbb{N} \to \mathbb{N}$ such that

$$(\forall(n), f(n) = 0 \vee f(n) = 1) \wedge (\chi_P(n) = 0 \Leftrightarrow \neg R(n)) \wedge (\chi_P(n) = 1 \Leftrightarrow R(n)).$$

Theorem 3.16. *The type of characteristic functions for a predicate P is a proposition, and is inhabited precisely if P is decidable.*

Proof. Fix characteristic functions f and g for R. We know $\big(f(n) = 0\big) + \big(f(n) = 1\big)$. In the first case, we have $\neg P(n)$ and so $g(n) = 0$. Otherwise, $f(n) = 1$ and so $P(n)$; hence, $g(n) = 1$.

If P has a characteristic function, then since $\mathbb{N}$ has decidable equality, P is decidable. Conversely, if P is decidable, then $i \circ p$ is a characteristic function, where $p : \mathbb{N} \to 2$ factors P. $\square$

3.5 Monads

A monad on a category $\mathcal{C}$ is a functor $T : \mathcal{C} \to \mathcal{C}$ equipped with natural transformations $\eta : \mathsf{id}_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ such that the following diagrams commute.

$$\begin{tikzcd}
T^3(X) \arrow[r, "\mu_{TX}"] \arrow[d, "T(\mu_X)"'] & T^2(X) \arrow[d, "\mu_X"] \\
T^2(X) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}
\qquad
\begin{tikzcd}
T^2(X) \arrow[dr, "\mu_X"'] & T(X) \arrow[l, "\eta_{T(X)}"'] \arrow[r, "T(\eta_X)"] \arrow[d, equal] & T^2(X) \arrow[dl, "\mu_X"] \\
& T(X) &
\end{tikzcd}$$

A monad T can equivalently be presented as a *Kleisli triple*: an operation T on objects, together with a family of maps $\eta_X : X \to T(X)$ and a *Kleisli extension* operator which takes a map $f : X \to T(Y)$ to a map $f^\sharp : T(X) \to T(Y)$, satisfying the *Kleisli laws*:

$$\begin{aligned}
\eta_X^\sharp &= \mathsf{id}_{T(X)} \\
f &= f^\sharp \circ \eta \\
(g^\sharp \circ f)^\sharp &= (g^\sharp) \circ (f^\sharp).
\end{aligned}$$

Given η and $(-)^\sharp$ we get the application on morphisms $f : X \to Y$ as

$$T(f) \stackrel{\mathsf{def}}{=} (\eta_Y \circ f)^\sharp$$

and $\mu : T^2 \to T$ as

$$\mu_X \stackrel{\mathsf{def}}{=} \mathsf{id}^\sharp_{T(X)} .$$

Functoriality of T, naturality of η and μ and the monad laws follow from the Kleisli equations. Conversely, given a monad (T, η, μ) and $f : X \to Y$ in $\mathcal{C}$, define $f^\sharp \stackrel{\mathsf{def}}{=} \mu_Y \circ T(f)$. Again, the Kleisli laws follow from the monad laws and naturality of η and μ.

Given a Kleisli triple $(T, \eta, (-)^\sharp)$, we can define *Kleisli composition* of $f : X \to T(Y)$ and $g : Y \to T(Z)$ by

$$g \,\square\, f \stackrel{\mathsf{def}}{=} (g^\sharp) \circ f.$$

We may internalize the definition of a monad by looking at type operators.

Definition 3.17. A *monad* is a map $T : \mathcal{U} \to \mathcal{U}$ together with a family of maps

$$\eta : \Pi(X : \mathcal{U}), X \to T(X)$$

and a Kleisli extension operator

$$(-)^\sharp : \Pi(X, Y : \mathcal{U}), (X \to TY) \to (TX \to TY)$$

such that

$$\begin{aligned} \eta^\sharp_X &= \mathsf{id}_{T(X)} \\ f &= f^\sharp \circ \eta \\ (g^\sharp \circ f)^\sharp &= (g^\sharp) \circ (f^\sharp) \end{aligned}$$

Definition 3.18. If η and $(-)^\sharp$ give $T : \mathcal{U} \to \mathcal{U}$ the structure of a monad, a *submonad* of T is an operation $S : \mathcal{U} \to \mathcal{U}$ such that for every X there is an embedding $i_X : SX \to TX$ such that

- for every $X : \mathcal{U}$, η_X factors through i_X, and
- the embedding i commutes with Kleisli extension.

That is, we have $\eta' : X \to SX$ and for every $f : X \to SY$, we have $f^{\sharp'} : SX \to SY$ such that the following diagram commutes:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\eta'"] \arrow[dd, bend right, "\eta"'] & SY \arrow[d, "i"] \\
SX \arrow[ur, "f^{\sharp'}"'] \arrow[d, "i"] & TY \\
TX \arrow[ur, "(i \circ f)^\sharp"'] &
\end{tikzcd}$$

We will be interested in the *lifting* monads $\mathcal{L}$ of Chapter 5. However, they are formed via quantification over a universe $\mathcal{U}$, and so can raise the universe level; naively, we have the lifting as a map $\mathcal{L} : \mathcal{U}_0 \to \mathcal{U}_1$; We discuss this discrepancy in Section 5.6, where we see that in fact lifting can be applied at higher universes, and there has type $\mathcal{U}_i \to \mathcal{U}_i$, so indeed forms a monad as stated above. However, size issues can also be resolved in another way, by making the definition of monad universe polymorphic: we can take a monad to be a map $T : \mathcal{U} \to \mathcal{V}$, for two universes $\mathcal{U}$ and $\mathcal{V}$, and then we have $\eta : \Pi(X : \mathcal{U}), X \to T(X)$, where then the type of η lives in universe $\mathsf{max}(\mathcal{U}', \mathcal{V})$, where $\mathcal{U}'$ is the universe above $\mathcal{U}$, as does the type of the Kleisli extension operator $(-)^\sharp : \Pi(X, Y : \mathcal{U}), : (X \to T(Y)) \to T(X) \to T(Y)$.

3.6 Choice principles

The forward direction of the following equivalence is often called the *type theoretic axiom of choice*.

Theorem 3.19. *For any $A, B : \mathcal{U}$ and $R : A \to B \to \mathcal{U}$, we have an equivalence,*

$$\Big(\Pi(x : A), \Sigma(y : B), R(x, y)\Big) \simeq \Big(\Sigma(f : A \to B), \Pi(x : A), R(x, f(x))\Big)$$

Proof. Let

$$P \overset{\mathsf{def}}{=} \Pi(x : A), \Sigma(y : B), R(x, y),$$

and

$$Q \overset{\mathsf{def}}{=} \Sigma(f : A \to B), \Pi(x : A), R(x, f(x)),$$

so that we are looking for an equivalence $P \simeq Q$. Define $h : P \to Q$ by

$$h(\psi) \overset{\mathsf{def}}{=} \big(\lambda x.\, \mathsf{pr}_0(\psi\, x), \lambda x.\, \mathsf{pr}_1(\psi\, x)\big)$$

and define $g : Q \to P$ by

$$g(f, \varphi) \overset{\mathsf{def}}{=} \lambda x.\big(f(x), \varphi(x)\big).$$

Then we have

$$g(h(\psi)) = g\big(\lambda x.\, \mathsf{pr}_0(\psi\, x), \lambda x.\, \mathsf{pr}_1(\psi\, x)\big) = \lambda x.\big(\, \mathsf{pr}_0(\psi\, x), \mathsf{pr}_1(\psi\, x)\big) = \lambda x.\psi(x) = \psi,$$

and

$$h(g(f, \varphi)) = h(\lambda x.(f(x), \varphi(x))) = (\lambda x.f(x), \lambda x.\varphi(x)) = (f, \varphi).$$

So h and g are inverse. $\square$

It is somewhat misleading to call this "the axiom of choice" as the notion of existence given by Σ is too strong for this to properly interpret the axiom of choice: Σ asserts a given witness, which we can then extract in a functional way. It is more in line with the usual uses of choice to instead weaken Σ to exists $\exists$. However, asserting this for all types and type families is too strong [86, Lemma 3.8.5], so we can only assert the axiom of choice for sets.

Axiom 3.20. The *axiom of choice* posits that given sets A and B and a relation $R : A \to B \to \mathcal{U}$ such that $R(x, y)$ is a proposition for all $x : A$ and $y : B$, there is an inhabitant of the type

$$\Big(\Pi(x : A), \exists(y : B), R(x, y)\Big) \to \Big(\exists(f : A \to B), \Pi(x : A), R(x, f(x))\Big).$$

We will be interested in various weakenings of the axiom of choice. For simplicity of notation, we will use an alternate characterization of the axiom of choice for the statement of such principles.

Lemma 3.21. *The axiom of choice is equivalent to the following:*

For all sets X and families $Y : X \to \mathcal{U}$ such that $Y(x)$ is a set for all x, we have

$$\big(\Pi(x : X), \|Y(x)\|\big) \to \|\Pi(x : X), Y(x)\|$$

Proof. Fix a type $A : \mathcal{U}$ and take $X \stackrel{\mathsf{def}}{=} A$ so that for any $B : \mathcal{U}$ the map

$$(f, \varphi) \mapsto \lambda x.(f(x), \varphi(x))$$

defines an equivalence

$$\Big(\Sigma(f : A \to B), \Pi(x : A), R(x, f(x))\Big) \simeq \Big(\Pi(x : A), \Sigma(y : B), R(x, y)\Big).$$

So then setting $Y(x) \stackrel{\mathsf{def}}{=} \Sigma(y : B), R(x, y)$, we have the backward direction.

For the forward, direction, assume we have $Y : X \to \mathcal{U}$ and define $B = \Sigma(x : X), Y(x)$ and $R : X \to B \to \mathcal{U}$ by

$$R(x, y) \stackrel{\mathsf{def}}{=} \mathsf{pr}_0(y) = x.$$

Then we know for any $x : X$ that $\Sigma(y : B), R(x, y)$ is equivalent to $Y(x)$, and so the type $\Pi(x : X), \exists(y : B), R(x, y)$ is equivalent to $\Sigma(x : X), \|Y(x)\|$. So if we assume the latter is inhabited then so is the former. Then by the axiom of choice, we have

$$\|\Sigma(f : X \to B), \Pi(x : X), \mathsf{pr}_0(f(x)) = x\|\,.$$

Then we have $\|\Pi(x : X), , Y(x)\|$ by functoriality of truncation (Lemma 2.14) using the function

$$\lambda(f, \varphi).\lambda x.\,\mathsf{transport}(\varphi(x), \mathsf{pr}_2(f(x))). \qquad \square$$

We will call statement of the form

$$\big(\Pi(x:X), \|Y(x)\|\big) \to \|\Pi(x:X), Y(x)\|$$

choice from X to Y, or *choice of Y over X*, and similarly if $P, Q : \mathsf{Set} \to \mathcal{U}$ are property of sets, we will call

$$P(X) \to \Big(\Pi(x:X), Q(Yx)\Big) \to \big(\Pi(x:X), \|Y(x)\|\big) \to \|\Pi(x:X), Y(x)\|$$

choice from P to Q. In this language, the full axiom of choice is choice from sets to sets. Two more crucial examples are *countable choice*, (choice from $\mathbb{N}$),

$$\big(\Pi(n:\mathbb{N}), \|Y(n)\|\big) \to \|\Pi(n:\mathbb{N}), Y(n)\|\,;$$

and *propositional choice*, (choice from propositions),

$$\mathsf{isProp}(X) \to \big(\Pi(x:X), \|Y(x)\|\big) \to \|\Pi(x:X), Y(x)\|\,.$$

Neither of these principles are provable in univalent type theory. Coquand et al. [22] recently gave a model (which validates univalence) but in which countable choice fails, and a similar model which invalidates a consequence of propositional choice. However, the *principle of unique choice* (choice from propositions to propositions) is provable in our type theory.

Theorem 3.22. *For any proposition X and any $P : X \to \mathcal{U}$ such that $\Pi(x:X), \mathsf{isProp}(Px)$ we have*

$$\big(\Pi(x:X), \|P(x)\|\big) \to \|\Pi(x:X), P(x)\|\,.$$

Proof. In fact, the assumptions give

$$\big(\Pi(x:X), \|P(x)\|\big) \to \Pi(x:X), P(x),$$

by Lemma 2.12. □

We also have choice from decidable propositions:

Theorem 3.23. *If P is a decidable proposition, then*

$$\big(\Pi(p : P), \|Q(p)\|\big) \to \|\Pi(p : P), Q(p)\| \, .$$

Proof. We do a case analysis on P: if $P = 0$, then we have $\Pi(p : P), Q(p)$, and so in particular,

$$\big(\Pi(p : P), \|Q(p)\|\big) \to \|\Pi(p : P), Q(p)\| \, .$$

If $P = 1$, then $\Pi(p : P), \|Q(p)\|$is equivalent to $\|Q(\star)\|$, and this in turn is equivalent to

$$\|\Pi(p : P), Q(p))\| \, .$$

□

3.7 Discussion

Lemma 3.13 has been discussed by Martín Escardó, and provides real-world examples of situations where $\|X\| \to X$. This is related to an interesting observation by Nicolai Kraus [48] that there is a dependent function f such that $f(|n|) = n$ for all $n : \mathbb{N}$, and to work by Kraus, Escardó and others [47, 49] on constancy and anonymous existence; Section 3.3 is also related to this work.

While the notion of monad we gave is good enough for our purposes, the universe is not a category but an ∞-category [92, 53, 88], although there is no definition of ∞-category in type theory at the time of writing. Monads in higher categories are somewhat more subtle, due to the coherence issues that arise. Some recent work by Riehl and Verity [70] is concerned with the issue of ∞-monads.

The axiom of choice has a subtle relationship with constructiveness. On one hand, you have that choice does not seem to correspond to an effective procedure in any way, and Diaconescu's theorem that choice implies excluded middle. On the other, you have Bishop's pronouncement that "a choice function exists in constructive mathematics, because a choice is implied by the very meaning of existence," [9] and the so-called *type-theoretic axiom of choice* which is a direct application of the definition of Σ and Π. A good overview of this situation is given by Martin-Löf [58]. The subtleties discussed by Martin-Löf seem to vanish in a univalent setting: reading the axiom of choice in the logic of structures (without truncations) gives the version that is “implied by the very meaning of existence”, while reading it in the logic of propositions (with truncations) gives the classical principle.

Part II

Partiality

We turn now to partiality. Classically, a partial function $X \rightharpoonup Y$ is taken to be a relation $R \subseteq X \times Y$ such that for each $x \in X$, there is at most one $y \in Y$ such that $R(x, y)$. While this approach can be used constructively, there are a few reasons we wish to avoid this representation of partial functions. Most importantly, expressing application of a partial function represented as a relation is cumbersome, awkward, and non-rigorous: We express the fact that there is at most one y such that $R(x, y)$ by saying that $\Sigma(y : Y), R(x, y)$ is a proposition. To apply a functional relation $R : X \times Y \to \mathcal{U}$ to $x : X$, we first demonstrate that $\Sigma(y : Y), R(x, y)$ is inhabited, and then appeal to the fact that it is a proposition to determine there is at most one, and then apply the second projection. In the case where f is not defined at x, the notation $f(x)$ does not even make sense since $\Sigma(y : Y), R(x, y)$ is empty.

In settings where functions take a more privileged position than in (e.g.) ZF, a partial function $X \rightharpoonup Y$ is often taken to be a (total) function $X \to Y + 1$, where the adjoined element represents an undefined value. This approach only works for partial functions $f : X \rightharpoonup Y$ where it is decidable whether f is defined on x.

We generalize the undefined value by replacing $Y + 1$ with a type of *partial elements* of Y. This can be expressed by taking a partial element to be a proposition (the extent of definition) together with a map from that proposition to Y (its value). In a classical setting, this type is equivalent to $Y + 1$, since a proposition is either true or false. A partial function $X \rightharpoonup Y$ is taken to be a total function from X to the type of partial elements of Y. We can extract the *value* of $f(x)$ (as an element of Y) when we know $f(x)$ to be a *total* element. However, we need to be able to lift data about elements of Y to data about partial elements of Y—in short, we need the operation taking Y to its type of partial elements to be a monad. This approach is directly related to synthetic domain theory, which we discuss in Section 4.3. One point made throughout the literature on synthetic domain theory is that the type of all partial elements (and the type of all partial functions) is often not very interesting; we instead want to focus on a class of particular partial functions and partial elements. Such a class can be defined by restricting our extents of definitions. The restricted partial elements operation succeeds in forming a monad precisely if the set of allowable extents of definition form a *dominance* (Definition 5.17). Many sets of propositions of interest arise by truncation from *structural dominances* (Definition 5.35), which are just Prop-indexed families of types closed under Σ such that the type indexed by 1 is inhabited. Unfortunately, sets of propositions arising in this way are dominances if and only if particular (weak) forms of choice hold (Theorem 5.32). Instead, we consider *lifted* functions arising by applying the same constructions directly to a structural dominance, instead of a dominance. The type of lifted functions arising in this way contains many representations of the same partial function, and while they do compose, there is no guarantee that this composition is associative. Nevertheless, we can tame these wild partial functions to get a notion of *disciplined map* (Definition 5.42). These maps do indeed compose (associatively), even in the absense of choice, and the information tracked explicitly by the lifted functions is now implicit.

Of particular interest are the *Rosolini structures* (a structural dominance) and the *Rosolini propositions* (their truncation), which are an abstract form of the *affirmable* or *semidecidable* truth values. This class of propositions are related to the extended naturals, $\mathbb{N}_\infty$ (defined by adding a point at infinity to $\mathbb{N}$), and to other approaches to partiality in type theory based on the delay monad (Section 4.1) and higher inductive-inductive types (Section 4.2). These will play a crucial role in Chapter 9, as they can be used as to define notions we expect to satisfy the hope of a notion of partial function which can be used as a surrogate for the computable functions.

This approach allows us to express application of partial functions directly. The statement that $f(x)$ is defined is a proposition about an existing partial-element, rather than a possibly non-existent element. There is another, less important benefit of the approach via partial elements: the full univalence axiom is required to identify functions with their graph (Theorem 3.6), and univalence appears repeatedly when trying to prove type families equal. The approach to partial functions as partial elements does not need univalence.

CHAPTER 4

Partiality in Type Theory and Topos Theory

Before developing our approach to partiality, we quickly review prior work on partiality in type theory. The standard approach to partiality in type theory center around the *delay monad*, which we define concretely using the extended natural numbers in Section 4.1. Kleisli maps for the delay monad can be viewed as partial functions equipped with intensional information. This extra intensional information means the delay monad does not work for our purposes: it makes partiality into structure, rather than property. Chapman, Uustalu and Veltri resolve this issue by quotienting the delay monad, but then countable choice is needed to show that the quotiented delay monad forms a monad—to show that partial functions compose. Our attempt to give partial functions via dominances in Chapter 5, leads us to the same place, so we discuss the details from the perspective of dominances there. More recently, Altenkirch, Danielsson and Kraus use a higher-inductive type to define the free ωCPO on a type X, and use Kleisli maps for this monad to give partial functions.

Our approach to partiality will center around a type-theoretic version of the notion of *dominance*. Dominances and many of the tools we use to work with dominances arise from synthetic domain theory and are also used in synthetic computability theory. Synthetic domain theory aims to develop domain theory by axiomatizing a particular topos of objects which naturally have dcpo structure. Synthetic computability theory is analogous. In Section 4.3, we give a brief overview of this work.

4.1 $\mathbb{N}_\infty$ and the delay monad

The *extended naturals*, $\mathbb{N}_\infty$, are of central importance in other approaches to partiality in type theory, and will be important in ours as well. The type is coinductively generated by

$$\mathsf{zero} : \mathbb{N}_\infty$$
$$\mathsf{succ} : \mathbb{N}_\infty \to \mathbb{N}_\infty$$

That is, $\mathbb{N}_\infty$ is the final coalgebra for $X \mapsto X + 1$. The coalgebra map is $\mathsf{pred} : \mathbb{N}_\infty \to \mathbb{N}_\infty + 1$, given by

$$\begin{aligned} \mathsf{pred}(\mathsf{zero}) &= \mathsf{inr}\,\star \\ \mathsf{pred}(\mathsf{succ}\,x) &= x \end{aligned}$$

As this type is coinductively generated, there is an element $\infty : \mathbb{N}_\infty$ satisfying the equation

$$\infty = \mathsf{succ}(\infty).$$

We cannot use the above coinductive definition, however, as we do not have coinduction in our type theory. We instead must give a concrete type which satisfies the required universal property. The type $\mathbb{N}_\infty$ should be thought of as the naturals extended by a point at infinity. As we cannot separate the extra point from $\mathbb{N}$, the type $\mathbb{N} + 1$ will not suffice. Instead, we we take $\mathbb{N}_\infty$ to be the type of binary sequences with at most one 1[30].

Definition 4.1. The *Cantor space* is the type $2^{\mathbb{N}}$.

Note that the cantor space can be seen both as the type of infinite binary sequences, and as the type of decidable predicates on $\mathbb{N}$.

Definition 4.2. For $\alpha : 2^{\mathbb{N}}$, define the type $\langle\alpha\rangle : \mathcal{U}$ by

$$\langle\alpha\rangle \overset{\mathsf{def}}{=} \Sigma(n : \mathbb{N}), \alpha_n = 1.$$

Define the *extended natural numbers* to be the type

$$\mathbb{N}_\infty \overset{\mathsf{def}}{=} \Sigma(\alpha : 2^{\mathbb{N}}), \mathsf{isProp}(\langle\alpha\rangle).$$

The element $\infty : \mathbb{N}_\infty$ is defined to be the constantly zero function,

$$\infty \stackrel{\mathsf{def}}{=} \lambda n.0.$$

Given $n : \mathbb{N}$, the sequence given by the decidable predicate $\lambda k.n = k$ only takes value 1 at input n, and so for each $n : \mathbb{N}$ there is an element $\overline{n} : \mathbb{N}_\infty$.

We could instead equivalently define $\mathbb{N}_\infty$ to be the set of *increasing* binary sequences.

Definition 4.3. A sequence $\alpha : 2^{\mathbb{N}}$ is *increasing* when

$$\Pi(n, m : \mathbb{N}), (\alpha_n = 1) \to (n < m) \to (\alpha_m = 1).$$

Lemma 4.4. *The type $\mathbb{N}_\infty$ is equivalent to the type of increasing binary sequences.*

Proof. Switch all bits after the first 1 in the sequence. Explicitly:

Let α be increasing. Define $\mu : \mathbb{N} \to \mathcal{U}$ by

$$\mu(n) \stackrel{\mathsf{def}}{=} (\alpha_n = 1) \times \Pi(k < n), \alpha_k = 0.$$

The right hand side is a decidable predicate by Lemma 3.13, and so μ factors through a map $p(\alpha) : \mathbb{N} \to 2$. Moreover, we know that the type

$$\Sigma(n : \mathbb{N}), (\alpha_n = 1) \times \Pi(k < n), \alpha_k = 0$$

is a proposition, and then so is $\langle p(\alpha) \rangle$.

Conversely, if μ is a sequence with at most one 1, we may define

$$i(\mu)_n = 1 \Leftrightarrow \exists(k \leq n), \mu_k = 1$$

Since $\mu_k = 1$ is decidable, we know that so is $\exists(k \leq n), \mu_k = 1$ by bounded search.

We have that for any increasing $\alpha : 2^{\mathbb{N}}$ that

$$i(p(\alpha))_n = 1 \Leftrightarrow \exists(k \leq n), p(\alpha)_k = 1.$$

Since α is an increasing sequence from 2, this is the same as

$$i(p(\alpha))_n = 1 \Leftrightarrow \alpha_n = 1.$$

Conversely, for any μ with at most one one, we have

$$p(i(\mu))_n = 1 \Leftrightarrow (i(\mu)_n = 1) \times \Pi(k < n), i(\mu)_k = 0,$$

which happens at the least n such that $\mu_n = 1$, but since $\langle\mu\rangle$ is a proposition, this is the unique value at which $\mu_n = 1$, and so

$$p(i(\mu))_n = 1 \Leftrightarrow (\mu_n = 1). \qquad \square$$

As usual with proposition-valued components in sums, we will often suppress the second component, proving that the required witness exists separately. We have the map $\mathsf{pred} : \mathbb{N}_\infty \to \mathbb{N}_\infty$ defined by

$$\begin{aligned} \mathsf{pred} &: \mathbb{N}_\infty \to \mathbb{N}_\infty \\ \mathsf{pred}(\alpha) &= (\lambda n.\alpha(n+1)). \end{aligned}$$

That $\langle \mathsf{pred}(\alpha)\rangle$ is a proposition when $\langle\alpha\rangle$ is follows from the fact that $\mathsf{pred}(\alpha)_n = 1 \Leftrightarrow \alpha_{n+1} = 1$.

Theorem 4.5. *The type $\mathbb{N}_\infty$ satisfies the correct universal property. That is, if $p : X \to X + 1$ for any type $X : \mathcal{U}$, then there is a unique $\varphi : X \to \mathbb{N}_\infty$ such that*

$$\mathsf{pred}(\varphi(x)) = (\varphi + 1)(p(x)).$$

Proof. The idea is to count how long it takes for p to give us $\mathsf{inr}\,\star$. To do this, first define the function $p' : X + 1 \to X + 1$,

$$\begin{aligned} p'(\mathsf{inl}\, x) &= px \\ p'(\mathsf{inr}\, \star) &= \mathsf{inr}\, \star, \end{aligned}$$

and then define $\varphi' : X + 1 \to \mathbb{N}_\infty$

$$\begin{aligned}
\varphi'(\mathsf{inl}\,x) &= 0 && \text{if } p'(\mathsf{inl}\,x) = \mathsf{inl}\,y \\
\varphi'(\mathsf{inl}\,x) &= 1 && \text{if } p'(\mathsf{inl}\,x) = \mathsf{inr}\,y \\
\varphi'(\mathsf{inr}\,\star) &= 0
\end{aligned}$$

Then $\varphi = \varphi' \circ p$.

It is easy to check that this is correct and unique. □

We have that $\mathbb{N}_\infty$ is a retract of $2^{\mathbb{N}}$: Define $f : 2^{\mathbb{N}} \to \mathbb{N}_\infty$ by

$$f(\alpha)(n) \stackrel{\mathsf{def}}{=} \begin{cases} \alpha(n) & \text{if } \alpha(k) = 0 \text{ for all } k < n \\ 0 & \text{otherwise} \end{cases}$$

Note that we are doing a case analysis on $\Pi(k < n), \alpha(k) = 0$, which is decidable by bounded search. We have that $\langle f(\alpha) \rangle$ is a proposition since $f(\alpha)(k) = 1$ exactly if k is the least such that $\alpha(k) = 1$. We call $f(\alpha)$ the *truncation* of α.

We immediately have,

Lemma 4.6. *The map $f : 2^{\mathbb{N}} \to \mathbb{N}_\infty$ above is a retraction of the inclusion $\mathbb{N}_\infty \to 2^{\mathbb{N}}$.*

Capretta defines, for any type X, the type $\mathsf{D}(X)$ *of delayed elements of X*, coinductively generated by

$$\begin{aligned}
&\mathsf{now} : X \to \mathsf{D}(X) \\
&\mathsf{delay} : \mathsf{D}(X) \to \mathsf{D}(X).
\end{aligned}$$

Again, we must define this type concretely in our type theory. Classically, the elements of $\mathsf{D}(X)$ can be written as $\mathsf{delay}^n(\mathsf{now}\,x)$, or delay^∞. Again, constructively, we cannot do a simple case analysis to determine which case is satisfied, but we may use the same trick as with $\mathbb{N}_\infty$.

Theorem 4.7. *The following types are equivalent:*

(i) $\Sigma(\mu : \mathbb{N}_\infty), \Pi(n : \mathbb{N}), X^{\mu = \overline{n}}$,

(ii) $\Sigma(\alpha : \mathbb{N} \to (X + 1)), \mathsf{isProp}\,\langle \alpha \rangle$,
where $\langle \alpha \rangle \stackrel{\mathsf{def}}{=} \Sigma(n : \mathbb{N}), \Sigma(x : X), \alpha(n) = \mathsf{inl}(x)$,

Proof. For $(\mu, \chi) : (i)$, we know that $(\mu = \overline{n}) \simeq (\mu_n = 1)$, which is a decidable predicate. Hence we can take

$$\alpha(n) = \begin{cases} \mathsf{inl}\,\chi(n, -) & \text{if } \mu = \overline{n}, \\ \mathsf{inr}(\star) & \text{otherwise.} \end{cases}$$

As there is at most one n with $\mu = \overline{n}$, we know that there is at most one n which takes the form $\mathsf{inl}\,x$.

Conversely, let $(\alpha, -) : (ii)$, and define

$$\mu(n) = \begin{cases} 1 & \text{if } \alpha(n) = \mathsf{inl}\,x, \\ 0 & \text{otherwise.} \end{cases}$$

And define $\varphi(n) = \lambda y.x$. $\square$

We will tacitly switch between these two representations of $\mathsf{D}\,X$, but we make (ii) official. In particular, we define

$$\begin{aligned} \mathsf{now}(x)(0) &= \mathsf{inl}\,x \\ \mathsf{now}(x)(n+1) &= \mathsf{inr}\,\star, \end{aligned}$$

and

$$\begin{aligned} \mathsf{delay}(\mu)(0) &= \mathsf{inr}\,\star \\ \mathsf{delay}(\mu)(n+1) &= \mu(n). \end{aligned}$$

We want $\mathsf{D}\,X$ to be the final coalgebra for $Y \mapsto Y + X$, for which we define the coalgebra map $\mathsf{tick} : \mathsf{D}\,X \to (\mathsf{D}\,X + X)$, by

$$\begin{aligned} \mathsf{tick}(\mathsf{now}\,x) &= \mathsf{inr}\,x \\ \mathsf{tick}(\mathsf{delay}\,\mu) &= \mathsf{inl}\,\mu. \end{aligned}$$

The proof of the following theorem is identical to the case for $\mathbb{N}_\infty$.

Theorem 4.8. *Given any type Y and a coalgebra $t : Y \to Y + X$, there is a unique coalgebra homomorphism $\varphi : Y \to \mathsf{D}\,X$ such that for all $y : Y$*

$$\mathsf{tick}(\varphi y) = (\varphi + X)(ty),$$

where

$$(\varphi + X)(\mathsf{inl}\,y) = \mathsf{inl}(\varphi y)$$
$$(\varphi + X)(\mathsf{inr}\,x) = \mathsf{inr}\,x.$$

$\mathsf{now} : X \to \mathsf{D}\,X$ gives an obvious candidate unit for D. Given $f : X \to \mathsf{D}\,Y$, we have a Kleisli extension $f^\sharp : \mathsf{D}\,X \to \mathsf{D}\,Y$ with

$$f^\sharp(\mathsf{now}\,x) = fx$$
$$f^\sharp(\mathsf{delay}\,\mu) = \mathsf{delay}(f^\sharp\mu).$$

Explicitly, for $x : X$, we define $\eta : X \to \mathsf{D}\,X$ by

$$\eta(x)(0) = \mathsf{inl}\,x$$
$$\eta(x)(n+1) = \mathsf{inr}\,\star$$

and if $f : X \to \mathsf{D}(Y)$, we define $f^\sharp : \mathsf{D}\,X \to \mathsf{D}\,Y$ by

$$f^\sharp(\mu)(n) = \begin{cases} \mathsf{inr}\,\star & \text{if } \Pi(k \le n), \mu(k) = \mathsf{inr}\,\star \\ f(x)(n-k) & \text{if } \Sigma(k \le n), (\mu(k) = \mathsf{inl}\,x) \end{cases}$$

Then we have that

$$\Big(f^\sharp(\mu)(n) = \mathsf{inl}\,y\Big) \Leftrightarrow \Big(\Sigma(k \le n), \Sigma(x{:}X), (\mu(n) = \mathsf{inl}\,x) \times f(x)(n-k) = \mathsf{inl}\,y\Big).$$

That is, $f^\sharp(\mu)$ takes the value $\mathsf{inr}\,\star$ for as long as μ does, and then takes the value of $f(x)$, where x is the unique value (if such exists) where $\mu(n) = \mathsf{inl}\,x$ for some n.

Theorem 4.9. *The maps* now *and* $(-)^\sharp$ *give* D *the structure of a monad.*

Proof. Rather than giving the (probably simpler) coinductive proof, we argue concretely. As $\eta(x)(0) = \mathsf{inl}\, x$, we have that

$$\eta^\sharp(x)(n) = \eta(x)(n-0).$$

Similarly, we have

$$f^\sharp(\eta x)(n) = f(x).$$

Expanding the definition of Kleisli extension, we see that $(g^\sharp f)^\sharp(\mu)(n) = \mathsf{inl}\, z$ precisely when

$$\Sigma(k \leq n), \Sigma(x{:}X), (\mu(k) = \mathsf{inl}\, x) \times \Sigma(j \leq n-k), \Sigma(y{:}Y), \big(f(x)(j) = \mathsf{inl}\, y\big) \times \big(g(y)(n-k-j) = \mathsf{inl}\, z\big).$$

Call this type A. Pictorially, we have that $(g^\sharp f)^\sharp(\mu)$ is

$$\underbrace{\star, \dots, \star,}_{\mu}\ \underbrace{\star, \star, \dots, \star,}_{f(x) \text{ where } \mu_k = x}\ \underbrace{\star, \dots, \star, z, \star, \dots, \star}_{g(y) \text{ where } f(x)_j = y}.$$

In words: $g^\sharp f^\sharp(\mu)$ is composed of a sequence of $\mathsf{inr}\, \star$ with possibly one element of the form $\mathsf{inl}\, z$. The terms of the sequence come from μ, until it takes value $\mathsf{inl}\, x$ (for some x), and then from $f(x)$, until it takes value $\mathsf{inl}\, y$ (for some y), and finally from $g(y)$ thereafter.

Similarly, we have that $g^\sharp(f^\sharp\mu)(n) = \mathsf{inl}\, z$ precisely if

$$\Sigma(k' \leq n), \Sigma(y : Y), \Sigma(j' \leq k'), \Sigma(x : X), (\mu(j') = \mathsf{inl}\, x) \times ((fx)(k'-j') = \mathsf{inl}\, y) \times (g(y)(n-k') = \mathsf{inl}\, z).$$

Call this type B. Pictorially, we have that $g^\sharp(f^\sharp\mu)(n) = \mathsf{inl}\, z$ is

$$\underbrace{\star, \dots, \star,}_{\mu}\ \underbrace{\star, \star, \dots, \star,}_{f(x) \text{ where } \mu_j = x}\ \underbrace{\star, \dots, \star, z, \star, \dots, \star}_{g(y) \text{ where } f(x)_k = y}.$$

We see that these two types are equivalent, by manipulating A as follows: First move all quantifiers to the front and rearrange, so that we have

$$\Sigma(k \leq n), \Sigma(y{:}Y), \Sigma(j \leq n-k), \Sigma(x{:}X), (\mu(k) = \mathsf{inl}\, x) \times \big(f(x)(j) = \mathsf{inl}\, y\big) \times \big(g(y)(n{-}k{-}j) = \mathsf{inl}\, z\big).$$

Next, given $k \leq n$ and $j \leq n-k$, define $j' = k$ and $k' = j + k$, so that $j' \leq k'$ and $k' \leq n$. So then j' and k' are of the correct form to satisfy B. That is, we have the map $A \to B$ defined by

$$(k, x, -, j, y, -, -) \mapsto (j + k, y, k, x, -, -, -),$$

with inverse

$$(k', y, j', x, -, -, -) \mapsto (j', x, -, k' - j', y, -, -).$$

So then $g^\sharp(f^\sharp\mu)(n) \Leftrightarrow (g^\sharp f)^\sharp(\mu)(n)$. Applying function and proposition extensionality, we get the Kleisli law. $\square$

The goal of the delay monad is to provide a satisfying account of partiality in type theory; however, the delay monad tracks intensional information—the timer given by delay—which makes equality on $\mathsf{D}(X)$ too strict. The obvious solution is to attempt to quotient $\mathsf{D}(X)$ by some equivalence relation, which is called *weak bisimilarity*. Capretta showed that the quotient of the delay monad by weak bisimilarity gives a monad in the category of setoids [14]. Later, Chapman, Uustalu and Veltri [16] show that this quotient gives a monad in a type theory with quotient inductive types assuming countable choice, so it seems that Capretta's result relies on the fact that countable choice is satisfied by the setoid model.

Weak bisimilarity is defined via a relation $- \downarrow - : DX \to X \to \mathcal{U}$ of *convergence*, defined inductively by $\mathsf{now}(x) \downarrow x$ and if $d \downarrow x$, then $\mathsf{delay}(d) \downarrow x$. We then define weak bisimilarity coinductively: if $d_1 \downarrow x$ and $d_2 \downarrow x$, then $d_1 \approx d_2$ and if $d_1 \approx d_2$, then $\mathsf{delay}(d_1) \approx \mathsf{delay}(d_2)$.

We can define both convergence and weak bisimilarity concretely on our representation of $\mathsf{D}\,X$.

Definition 4.10. For $d : \mathsf{D}\,X$ and $x : X$, we say that d *converges to* x if x is the unique value such that $d(n) = \mathsf{inl}\,x$. That is,

$$d \downarrow x \stackrel{\mathsf{def}}{=} \Sigma(n : \mathbb{N}), \big(d(n) = \mathsf{inl}\,x\big).$$

We say that elements d_1 and d_2 of $\mathsf{D}\,X$ are *weakly bisimilar* if for all $x : X$ we have $d_1 \downarrow x$ iff $d_2 \downarrow x$. That is

$$(d_1 \approx d_2) \stackrel{\mathsf{def}}{=} \Pi(x : X), (d_1 \downarrow x) \Leftrightarrow (d_2 \downarrow x).$$

Following [16], let $\overline{\mathsf{D}}(X) \stackrel{\mathsf{def}}{=} \mathsf{D}(X)/{\approx}$. They proved the following.

Theorem 4.11 ([16], Section 7)**.** *Assuming countable choice,* $\overline{\mathsf{D}}(X)$ *is a monad.*

We will see in Section 5.7 that $\overline{\mathsf{D}}(X)$ is equivalent to our *Rosolini lifting*, and we will subsequently discuss weakenings of countable choice that are sufficient to show that $\overline{\mathsf{D}}(X)$ is a monad, culminating in a necessary and sufficient weakening, Theorem 5.30.

4.2 Partiality via higher-inductive types

Altenkirch, Danielsson and Kraus instead define a notion of partiality by giving a higher inductive-inductive type [2]—simultaneous defining a higher inductive type $X_\perp$ and a binary relation $\sqsubset$ on $X_\perp$. Their goal was to give an extensional version of the delay monad which was a monad even without countable choice. The type $X_\perp$ is essentially the free ωCPO over X.

Definition 4.12. For each type X define simultaneous the type $X_\perp : \mathcal{U}$ and a binary relation $\sqsubseteq : X_\perp \to X_\perp \to \mathcal{U}$ inductively with constructors for $X_\perp$

- a map $\eta : X \to X_\perp$,
- an element $\perp : X_\perp$,
- for each $s : \mathbb{N} \to X_\perp$ and witness $p : \Pi(n : \mathbb{N}), s_n \sqsubseteq s_{n+1}$, an element $\sqcup(s, p) : X_\perp$,
- a path constructor $(x \sqsubseteq y) \to (y \sqsubseteq x) \to x = y$, and
- a set truncation $\Pi(x, y : X_\perp), \Pi(p, q : x = y), p = q$;

and constructors for $\sqsubseteq$

- $x \sqsubseteq x$ for each $x : X_\perp$,

- if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$,
- $\bot \sqsubseteq x$ for each $x : X_\bot$,
- for each $n : \mathbb{N}$ we have $s_n \sqsubseteq \sqcup(s, p)$,
- if $\Pi(n : \mathbb{N}), s_n \sqsubseteq x$ then $(\sqcup(s, p) \sqsubseteq x)$.

In the paper, they show that $X_\bot$ behaves like the lifting in classical domain theory, in the sense that $X_\bot$ is the initial ωCPO that X maps into (Theorem 2) and moreover η is injective, $\eta(x)$ is maximal with respect to $\sqsubseteq$ for each $x : X$, and $\eta(x) \neq \bot$ for all $x : X$ (Corollary 8).

4.3 Synthetic domain theory and synthetic computability theory

Domain theory arose from a problem in denotational semantics: Languages with non-termination cannot be directly modeled in the category of sets, since some functions are partial. Since non-termination behaves in non-trivial ways, there are limits on how functions can interact with the undefined value. To resolve this, Dana Scott [76] introduced *domains*, directed- or ω-complete partial orders with bottom. The bottom element represents an undefined value, and maps between them are required to be monotone and preserve directed suprema. This structure effectively captures the behavior of non-termination. However, Scott wondered whether there was some alternative axiomatization of sets in which sets naturally came equipped with the required structure. In his PhD thesis [75], Giuseppe Rosolini took up this question, giving a way to access categories of partial maps from within a category. The basic idea, which we will explore in more depth from a type-theoretic perspective in Chapter 5, is to isolate a subset $\Sigma \subseteq \Omega$ of truth values in a topos, and consider the maps $X \to Y_\bot$, where $Y_\bot$ is the set

$$Y_\bot = \{A \in \Omega^Y \mid (\forall x, y.A(x) \to A(y) \to x = y) \wedge (\exists x.A(x) \in \Sigma)\}.$$

In other words, $Y_\bot$ is the set of subsingleton subsets of Y whose *extent of definition* is in Σ. In order for these Σ*-partial functions* to contain total functions, we need the true proposition to be in Σ. Moreover, in order for the Σ-partial functions to form a category, we need to impose an additional restriction that for all propositions $p, q \in \Omega$,

$$p \in \Sigma \wedge (p \to (q \in \Sigma)) \to ((p \wedge q) \in \Sigma).$$

The two conditions make Σ into a *dominance*.

Rosolini's approach was concrete: while he considered the internal language of a topos, he worked with specific toposes in an informal metamathematics; in particular, he examined the effective and recursive toposes, and considered *effective objects* in a topos. However, his work allowed synthetic axiomatizations, which were pursued by a number of authors studying *synthetic domain theory* (See, for example [44, 89, 68]). The idea is to pursue Scott's idea of an axiomatization of a category of domains.

Since dominances are a way to approach partial functions and effectiveness, the notion features in Bauer's [5, 6] approach to synthetic computability. Here, Bauer is axiomatizing a category of sets (the *modest sets* [43]) arising from realizability toposes.

In both synthetic domain theory and synthetic computability theory, the dominance of interest is a dominance Σ representing the semidecidable propositions, introduced also by Rosolini. The dominance is given by

$$p \in \Sigma \Leftrightarrow \exists(\alpha : \mathbb{N} \to 2), p \leftrightarrow \big(\exists(n : \mathbb{N}), \alpha(n) = 1\big).$$

That is, the propositions in Σ are those which, when true, are observably true.

Most approaches to synthetic domain theory are topos-theoretic in nature, but it is worth pointing to two (Reus and Streicher [68] and Reus [69]) that use a type-theoretic axiomatization. We will briefly describe the Approach by Reus here, and compare this approach to synthetic domain theory to our approach to partiality in Section 5.11. Reus uses a version of the calculus of construction with an impredicative universe of propositions, which are assumed to be subsingletons (i.e., propositions in our sense), but note that there it is not required that all subsingletons are propositions. Reus also assumes a universe of *sets* (which need not be sets in our sense). Reus assumes function extensionality, but not proposition extensionality, which is not needed in the development there. Additionally, in contrast to a univalent setting, the principle of unique choice is not available in the calculus of constructions, so it is assumed explicitly in Reus's development. Rather than giving a dominance, Reus's axiomatization of Σ includes a number of principles that relate Σ to domain theory, without reference to partial functions. We explain their axiomatization in Section 5.11, where we have more context for understanding it, but note that they include a version of Markov's Principle (see Section 1.3), as well as a version of Phoa's Principle [83], both of which are not available in our setting.

4.4 Discussion

The notion of partiality given by the delay monad is intensional. In particular, a single partial function $f : X \rightharpoonup Y$ has many representations as a function $F : X \to \mathsf{D}(Y)$, based on how many applications of delay occur in the output of F. Quotienting by bisimilarity resolves this issue at the cost of composability of partial functions, which does not seem like a worthwhile cost.

The approach using quotient-inductive-inductive types resolves the issue without the cost, but it doesn't seem to work for the purposes of Part III: we want it to be possible to use our partial functions as an abstract representation of the computable functions. Our first attempt (the Rosolini partial functions of Chapter 5) is too big for this—we will instead consider a subtype of the Rosolini partial functions, the *disciplined maps* of Section 5.10—but the QIIT approach expands the set of Rosolini partial functions (compare Lemmas 13 and 14 from [2] with our Theorem 5.28).

The Kleisli category for the delay monad is morally an intensional version of the category of Σ-partial maps from synthetic domain theory. The approach via dominances puts this notion of partiality into a more general context; we transfer the relevant parts of this context to the univalent setting in Chapter 5. In the synthetic approach, however, The core categories of interest is the effective topos, and its category of *effective objects*. The effective topos has enough structure that we can show in its internal logic that Σ is indeed a dominance, as well as enough structure that we can show externally that the Σ-partial functions capture the functions of interest (the computable or continuous functions). In the more general situation that we consider, neither of these will be the case. Chapter 5 is concerned with the problem of showing a set of propositions to be a dominance, while Chapter 9 is concerned with the computability of maps arising from our version of Σ, which we call the set of *Rosolini propositions*.

CHAPTER 5

Partiality in Univalent Mathematics

We now turn in earnest to the notion of partial function. Recall that a motivating goal is to find a notion of partial function for which it is consistent that all such partial functions are computable. We expect such partial functions to include all total functions, and to be composable, even in the absence of countable choice and Markov's principle. A great deal of this chapter is concerned with attempting to satisfy those requirements.

In Sections 5.1-5.4, we examine the class of all partial functions. In Sections 5.5-5.9 we attempt to restrict the class of partial functions using *dominances* (Definition 5.17)—subsets of Prop satisfying certain closure conditions which ensure that the partial functions just defined are composable (see Theorems 5.20 and 5.21). We examine in particular the Σ_1 propositions, which we call *Rosolini* propositions, and relate this to prior work on partiality. A weakening of countable choice is required to show that the partial functions arising from Rosolini propositions are closed under composition (see Theorems 5.29, 5.30, and 5.33). This fact generalizes (Theorem 5.32) to any dominance arising by truncation from *structural dominances* (Definition 5.35). The consequence is that some amount of choice is required to restrict partial functions via dominances. In Section 5.10 we give a different approach using what we call *disciplined maps*. This approach still builds off of the theory of dominances. Notably, the disciplined maps can be shown to compose without countable choice, and in the presence of countable choice, the disciplined maps are exactly the Rosolini partial functions. In Part II, we will discuss the relevance of this.

5.1 Single-valued relations

We saw in Theorem 3.6 that assuming univalence we have an equivalence

$$(X \to Y) \simeq (\Sigma(R : X \to Y \to \mathcal{U}), \Pi(x : X), \mathsf{isContr}(\Sigma(y : Y), R(x, y))).$$

That is, we saw that a function is relation with a unique value for each $x : X$, matching the classical intuition. Classically, the partial functions arise by considering instead relations for which there is instead at most one $y : Y$ with $R(x, y)$ for each $x : X$.

Definition 5.1. A relation $R : X \to Y \to \mathcal{U}$ is *single-valued* when

$$\Pi(x : X), \mathsf{isProp}(\Sigma(y : Y), R(x, y)).$$

Note that being single-valued is a proposition. As with the definition of functional relation in Section 3.2, the more traditional definition doesn't work in general. In this case, the traditional definition would say that a relation is single-valued when

$$\Pi(x : X), \Pi(y, y' : Y), R(x, y) \to R(x, y') \to y = y'.$$

Again, if Y is not a set, then $y = y'$ may have many witnesses, and so the above type may have interesting structure. Instead, we want to say that a relation R is single-valued not only when all $y : Y$ such that $R(x, y)$ are equal, but when there is at most one pair (y, r) such that $r : R(x, y)$. In the case where Y is a set, the traditional definition is equivalent to the definition we give.

There is another traditional approach to the definition of partial function, more common in category theoretic settings. A partial function $X \rightharpoonup Y$ is a function $f : A \to Y$ for some A which embeds into X. We can then represent partial functions $X \rightharpoonup Y$ via two separate types in the next universe: via the type

$$\Pi(x : X), \mathsf{isProp}(\Sigma(y : Y), R(x, y)), \tag{5.1}$$

or via the type

$$\Sigma(A : \mathcal{U}), (e : A \to X), \mathsf{isEmbedding}(e) \times (A \to Y). \tag{5.2}$$

We have maps between them as follows: Given a single-valued relation $R : X \to Y \to \mathcal{U}$, define $A : \mathcal{U}$ by

$$A \stackrel{\mathsf{def}}{=} \Sigma(x : X), \Sigma(y : Y), R(x, y),$$

with $e : A \to X$ the first projection and $f : A \to Y$ the second projection. We need to see that for any $x : X$ the type

$$\Sigma((x', y, r) : \Sigma(x : X), (y' : Y), R(x', y)), x' = x$$

is a proposition. By Lemma 1.24, this type is equivalent to

$$\Sigma(y : Y), R(x, y),$$

which is a proposition since R is single-valued. This gives us a function $F : (5.1) \to (5.2)$,

$$F(R, -) \stackrel{\mathsf{def}}{=} (\Sigma(x : X), \Sigma(y : Y), R(x, y), \mathsf{pr}_0, -, \mathsf{pr}_1).$$

Conversely, given an embedding $e : A \hookrightarrow X$ with a map $f : A \to Y$, define $R : X \to Y \to \mathcal{U}$ by

$$R(x, y) \stackrel{\mathsf{def}}{=} \Sigma(a : A), (e(a) = x) \times (f(a) = y).$$

We need to see that for any x, the type

$$\Sigma(y : Y), \Sigma(a : A), (e(a) = x) \times (f(a) = y)$$

is a proposition. By reshuffling, this type is equivalent to

$$\Sigma(a : A), (e(a) = x) \times \Sigma(y : Y), f(a) = y,$$

and since $\Sigma(y : Y), f(a) = y$ is the singleton at $f(a)$, this type is in turn equivalent to

$$\Sigma(a : A), e(a) = x,$$

which is a proposition since e is an embedding. This gives us a function $G : (5.2) \to (5.1)$,

$$G(A, e, -, f) = \Big(\big(\Sigma(a : A), (e(a) = x) \times (f(a) = y)\big), -\Big).$$

We want G and F to be equivalences, but they are only equivalences up to equivalence. That is, we have the following

Lemma 5.2. *For any single-valued $R : X \to Y \to \mathcal{U}$, we have that the first projection of $G(F(R, -))$ is equivalent to R. Similarly, For any $e : A \hookrightarrow X$ and $f : A \to Y$ we have an equivalence between the type $A' \stackrel{\mathsf{def}}{=} \mathsf{pr}_0(F(G(A)))$ and A. Hence, in the presence of univalence, the type of single-valued relations is equivalent to the type of functions from a subtype of X to Y.*

Proof. Let $R : X \to Y \to \mathcal{U}$ be single-valued. We need to see that for any $x : X$ and $y : Y$, the type

$$\Sigma((x', y', r)\Sigma(x' : X), (y' : Y), R(x, y)), x' = x \times y' = y$$

is equivalent to $R(x, y)$, which follows by two applications of Lemma 1.24.

Conversely, let $e : A \hookrightarrow X$ and $f : A \to Y$. We need to see that A is equivalent to the type

$$\Sigma(x : X), \Sigma(y : Y), \Sigma(a : A), \big(e(a) = x\big) \times \big(f(a) = y\big).$$

This type reshuffles to

$$\Sigma(a : A), (\Sigma(x : X), e(a) = x) \times (\Sigma(y : Y), f(a) = y),$$

which is a sum of contractible types over A. $\square$

5.2 Partial elements

We will take a third approach to partial functions, via partial elements. By analogy with the equivalence $Y \simeq (1 \to Y)$, we expect a partial element of A to correspond to a partial function $1 \rightharpoonup A$. Taking partial functions as relations, this would give us the type

$$\Sigma(A : Y \to \mathcal{U}), \mathsf{isProp}(\Sigma(y : Y), Ay).$$

We could instead define partial elements directly as follows.

Definition 5.3. For a type X, the *lift* of X, or the type of *partial elements* of X, is the type

$$\mathcal{L}(X) \stackrel{\mathsf{def}}{=} \Sigma(P : \mathcal{U}), \mathsf{isProp}(P) \times (P \to X).$$

The first component of a partial element is called its *extent of definition* and the third component is its *value* ; so we have maps

$$\begin{aligned} \mathsf{defined} &: \mathcal{L}Y \to \mathcal{U}, \\ \mathsf{value} &: \Pi(u : \mathcal{L}Y), \mathsf{defined}\, u \to Y. \end{aligned}$$

A partial element $y : \mathcal{L}Y$ is *defined* or *total* if $\mathsf{defined}(y)$. For any type Y, there is a unique *undefined* element $\bot : \mathcal{L}Y$ with extent of definition $\emptyset$.

A *partial function from X to Y* is a function $f : X \to \mathcal{L}Y$. If $f : X \to \mathcal{L}Y$, we will sometimes write $f_e \stackrel{\mathsf{def}}{=} \mathsf{defined} \circ f$ and $f_v \stackrel{\mathsf{def}}{=} \mathsf{value} \circ f$.

To see the comparison with partial elements as partial functions from 1, write

$$\mathcal{L}'(Y) \stackrel{\mathsf{def}}{=} \Sigma(A : Y \to \mathcal{U}), \mathsf{isProp}(\Sigma(y : Y), A(y)).$$

We have a map

$$\begin{aligned} \mathsf{rel} : \mathcal{L}(Y) &\to \mathcal{L}'(Y) \\ \mathsf{rel}(P, w, \varphi) &\stackrel{\mathsf{def}}{=} \big((\lambda y.\Sigma(p : P), \varphi(p) = y)\,,\, w\big) \end{aligned}$$

where $w : \mathsf{isProp}(\Sigma(y : Y), (p : P), \varphi(p) = y)$ is constructed by observing that this is equivalent to $\mathsf{isProp}(\Sigma(p : P), (y : Y), \varphi(p) = y)$, and this is a sum of a contractible type over a proposition. Similarly, we have

$$\begin{aligned} \mathsf{ele} &: \mathcal{L}'(Y) \to \mathcal{L}(Y) \\ \mathsf{ele}(A, w) &\stackrel{\mathsf{def}}{=} \big((\Sigma(y : Y), A(y))\,,\ w\,,\ \mathsf{pr}_0\big). \end{aligned}$$

Lemma 5.4. rel *is a section of* ele*, and for any* $A : Y \to \mathcal{U}$ *with* $w : \mathsf{isProp}(\Sigma(y : Y), A(y))$ *and* $y : Y$ *the type* $A(y)$ *is equivalent to the first component of* $\mathsf{rel}(\mathsf{ele}(A, w))$ *applied to* y*.*

Proof. Let $(A, w) : \mathcal{L}'(Y)$. The relation defined by $\mathsf{rel}(\mathsf{ele}(A, w))$ is

$$R(y) = \sum_{(y',a):\Sigma(y':Y),A(y')} (y = y').$$

Re-associating, this is equivalent to

$$\Sigma(y' : Y), A(y') \times (y = y').$$

This in turn is equivalent to $A(y)$ by the map $(y', a', p) \mapsto \mathsf{transport}(p^{-1}, a')$ with inverse mapping a to (y, a, refl).

Now let $x \stackrel{\mathsf{def}}{=} (P, w, \varphi) : \mathcal{L}(Y)$. We need only see that the extent of definition of $\mathsf{ele}(\mathsf{rel}(x))$ is equivalent to P, since P is a proposition. We have

$$\mathsf{defined}(\mathsf{ele}(\mathsf{rel}(x))) = \Sigma(y : Y), (p : P), (\varphi(p) = y).$$

Re-associating as above, this type is equivalent to $P \times 1$. $\square$

Note that without univalence the equivalence

$$Ay \simeq \Sigma((y', a) : \Sigma(y' : Y), Ay'), (y' = y),$$

is insufficient to prove that ele is an equivalence, even in the presence of proposition extensionality. While $\Sigma(y : Y), Ay$ is a proposition, we don't know in general that $A(y)$ is. In the case that Y is a set, however, we can show that $A(y)$ must be a proposition, by Lemma 3.5. Then we have the following.

Lemma 5.5. *For any set Y, the functions* rel *and* ele *determine an equivalence between the types $\mathcal{L}(Y)$ and $\mathcal{L}'(Y)$.*

Proof. We have already seen that rel is a section of ele, so we need to see that ele is a section of rel. Fix $(A, w) : \mathcal{L}'(Y)$ By function extensionality, and since the second component of $\mathcal{L}'(Y)$ is a proposition, it suffices to turn the equivalence $A(y) \simeq \mathsf{pr}_0(\mathsf{rel}(\mathsf{ele}(A, w)))(y)$ into an equality. As Y is a set and $\Sigma(y : Y), A(y)$ is a proposition by assumption, we have by Lemma 3.5 that $A(y)$ is a proposition. Then since $A(y) \simeq \mathsf{pr}_0(\mathsf{rel}(\mathsf{ele}(A, w)))(y)$, we may apply proposition extensionality to get the desired equality $A(y) = \mathsf{pr}_0(\mathsf{rel}(\mathsf{ele}(A, w)))(y)$. $\square$

By considering maps into $\mathcal{L}'(Y)$ and $\mathcal{L}(Y)$, we can collect Lemmas 5.5 and 5.2 into the following theorem.

Theorem 5.6. *For any type X and any set Y, the following types are equivalent:*

(i) $\Sigma(R : X \to Y \to \mathcal{U}), \Pi(x : X), \mathsf{isProp}(\Sigma(y : Y), R(x, y))$

(ii) $\Sigma(A : \mathcal{U}), (e : A \to X), \mathsf{isEmbedding}(e) \times (A \to Y)$

(iii) $X \to \mathcal{L}(Y)$.

Moreover, in the presence of univalence, we may drop the condition that Y is a set.

We have an embedding

$$\eta : Y \to \mathcal{L}Y$$

given by

$$\eta(y) \stackrel{\mathsf{def}}{=} (1, -, \lambda u.y).$$

Lemma 5.7. *The map $\eta : Y \to \mathcal{L}Y$ is an embedding.*

Proof. Since $1 = 1$ is contractible, $\eta(x) = \eta(y)$ is equivalent to

$$\lambda u.x =_{(1 \to Y)} \lambda u.y.$$

By function extensionality and the induction principle for 1, this is equivalent to

$$(\lambda u.x)(\star) = (\lambda u.y)(\star),$$

which reduces to $x = y$. $\square$

Moreover, we can extend a function $f : X \to \mathcal{L}\,Y$ to a function $f^\sharp : \mathcal{L}\,X \to \mathcal{L}\,Y$ given by

$$\begin{aligned} f^\sharp(P : \mathcal{U}, -, \varphi : P \to X) \quad &\stackrel{\mathsf{def}}{=} \quad (Q : \mathcal{U}, -, \gamma : Q \to Y), \\ \text{where} \quad & Q \stackrel{\mathsf{def}}{=} \Sigma(p : P), \mathsf{defined}(f(\varphi(p))), \\ & \gamma(p, e) \stackrel{\mathsf{def}}{=} \mathsf{value}(f(\varphi(p)))(e), \end{aligned}$$

where the witness that Q is a proposition comes from the fact that propositions are closed under sums.

For predicativity reasons, $(\mathcal{L}, \eta, (-)^\sharp)$ does not actually give a monad, since $\mathcal{L}$ raises universe levels. We discuss these issues in 5.6, but we still prove the monad laws here: for our purposes, we only need $(-)^\sharp$ for composition of partial function, and η to give us the total elements.

Theorem 5.8. *The maps η and $(-)^\sharp$ give $\mathcal{L} : \mathcal{U} \to \mathcal{U}_1$ the structure of a monad.*

Proof. For notational convenience, we will completely suppress the witness that the extent of definition of a partial element is a proposition.

$$\begin{aligned} \eta^\sharp(P, \varphi) \quad &= \quad (\Sigma(p : P), \mathsf{defined}(\eta(\varphi p))\ ,\ \lambda(p, q).\, \mathsf{value}(\varphi p) q) \\ &= \quad ((\Sigma(p : P), 1), \lambda(p, q).\varphi p) \\ &= \quad (P \times 1, \lambda(p, q).\varphi p) \\ &= \quad (P, \varphi), \end{aligned}$$

where the last equality follows from proposition extensionality.

$$\begin{aligned} f^\sharp(\eta x) \quad &= \quad f^\sharp(1, \lambda p.x) \\ &= \quad (\Sigma(p : 1), \mathsf{defined}(f(x))\ ,\ \lambda(p, q).\, \mathsf{value}(f(x)) q) \\ &= \quad (\mathsf{defined}(f(x)), \mathsf{value}(f(x))) \\ &= \quad fx. \end{aligned}$$

Now let $g : Y \to \mathcal{L}\,Z$ and $f : X \to \mathcal{L}\,Y$. We compute

$$\begin{aligned}
(g^\sharp f)^\sharp(P,\varphi) &= \Big(\Sigma(p:P),(g^\sharp f)_e(\varphi p)\,,\ \lambda(p,q).(g^\sharp f)_v(\varphi p)q\Big)\\
&= \Big(\Sigma(p:P),g_e^\sharp f(\varphi p)\,,\ \lambda(p,q).g_v^\sharp(f(\varphi p))q\Big)\\
&= \Big(\Sigma(p:P),(q:f_e(\varphi p)),g_e(f_v(\varphi p)q)\,,\ \lambda(p,(q,r)).g_v(f_v(\varphi p)q)r\Big)\\
&= \Big(\Sigma((p,q):\Sigma(p:P),f_e(\varphi p)),g_e(f_v(\varphi p)q)\,,\ \lambda((p,q),r).g_v(f_v(\varphi p)q)r\Big)\\
&= g^\sharp\Big(\Sigma(p:P),f_e(\varphi p)\,,\ \lambda(p,q).f_v(\varphi p)q\Big)\\
&= g^\sharp(f^\sharp(P,\varphi)).
\end{aligned}$$

The first 3 equalities are just expansion of definitions; the third follows from the equivalence between $\Sigma(a:A),(b:Ba),C(a,b)$ and $\Sigma((a,b):\Sigma(a:A),B(a)),C(a,b)$, and the last two again apply definitions. □

This gives us a Kleisli composition operator we denote with "$\square$". That is, we have

$$g \,\square\, f \stackrel{\mathsf{def}}{=} g^\sharp \circ f.$$

We said that a partial element $p : \mathcal{L}\,Y$ is *defined* if $\mathsf{defined}(p)$. It would also be natural to say that p is defined if p is in the image of η. In fact, these are equivalent. Moreover, since η is an embedding, we know that $\mathsf{fib}_\eta(p)$ is a proposition, so we have

Theorem 5.9. *The following types are equivalent for any $p : \mathcal{L}\,Y$.*

1. $\mathsf{defined}(p)$,
2. $\exists(y:Y), p = \eta y$,
3. $\Sigma(y:Y), p = \eta y$.

Proof. The type $\Sigma(y:Y), p = \eta y$ is the fiber of η over p. As η is an embedding, it has propositional fibers, so $\Sigma(y:Y), p = \eta y$ is a proposition. Hence, we have an equivalence between $(\exists(y:Y), p = \eta y)$ and $\Sigma(y:Y), p = \eta y$.

As $\mathsf{defined}(p)$ is a proposition, we only need to see that $\mathsf{defined}(p) \Leftrightarrow \exists(y:Y), p = \eta y$.

Let p be defined so that $\mathsf{defined}(p) = 1$. Then we have by definition that $p = \eta(\mathsf{value}(p)(*))$. Conversely, if $\exists(y:Y), p = \eta y$, since $\mathsf{defined}(p)$ is a proposition, it is enough to define a map

$(\Sigma(y : Y), p = \eta y) \to \mathsf{defined}(p)$. If $p = \eta y$, then we have that $\mathsf{defined}(p) = \mathbb{1}$ by the characterization of equality in Σ types. $\square$

Definition 5.10. A partial function $f : X \to \mathcal{L}Y$ is *total* if it is defined everywhere:

$$\mathsf{total}(f) \stackrel{\mathsf{def}}{=} \Pi(x : X), \mathsf{defined}(f(x)).$$

A total function is a function $X \to Y$, and we expect the above notion to align with the notion of total function. Indeed, this is the case:

Lemma 5.11. *For any partial function $f : X \to \mathcal{L}Y$, the following types are equivalent:*

1. *the type of factorizations of f through Y:*

$$T(f) \stackrel{\mathsf{def}}{=} \Sigma(g : X \to Y), f = \eta \circ g.$$

2. *the type $D(f) \stackrel{\mathsf{def}}{=} \Pi(x : X), \mathsf{defined}(f(x))$.*

Proof. We have that $T(f)$ is equivalent by function extensionality to

$$T(f) \stackrel{\mathsf{def}}{=} \Sigma(g : X \to Y), \Pi(x : X), f(x) = \eta(g(x)).$$

Recall that by Theorem 3.19 we have for any type family R

$$\Big(\Sigma(g : X \to Y), \Pi(x : X), R(x, g(x))\Big) \simeq \Big(\Pi(x : X), \Sigma(y : Y), R(x, y)\Big).$$

In particular, for $R(x, y) \stackrel{\mathsf{def}}{=} f(x) = \eta(y)$, we have that $T(f)$ is equivalent to

$$\Pi(x : X), \Sigma(y : Y), f(x) = \eta(y).$$

By Theorem 5.9, this type is equivalent to $D(f)$. $\square$

Corollary 5.12. *There is at most one factorization of a partial function $f : X \to \mathcal{L}(Y)$ through a total function $g : X \to Y$. That is, the type $T(f)$ above is a proposition.*

The takeaway here is that from the knowledge that $f : X \to \mathcal{L}Y$ is defined everywhere, we can treat f as a function $X \to Y$. Justified by the above results, we will call f a *total function* both to mean that $f : X \to \mathcal{L}(Y)$ such that $\mathsf{total}(f)$ is inhabited, but also to emphasize that f is an ordinary function $f : X \to Y$.

5.3 Domain and range of partial functions

A function $f : X \to \mathcal{L}(Y)$ can be viewed either as an ordinary function from X to $\mathcal{L}(Y)$, or as a partial function from X to Y. When we view such a function as a partial function, we want the notion of image and domain to be different. If we view a partial function $X \rightharpoonup Y$ instead as a relation $R : X \to Y \to \mathcal{U}$, then the *range* of R is $\Sigma(y : Y), \exists(x : X), R(x, y)$, and the *domain* of R is $\Sigma(x : X), \exists(y : Y), R(x, y)$. Similarly, if we view a partial function $X \rightharpoonup Y$ as a function $f : A \to Y$ for a subtype A of X, then the range of f is $\Sigma(y : Y), \exists(a : A), f(a) = y$, and the domain is A. These notions align, and moreover, we can translate the notion to the view of partial functions as functions into a type of partial elements. To separate the image of a partial function from the image of an ordinary function, we stick to the name *range*. We will be more interested in the *predicates* for range and domain than the total types of these predicates.

Definition 5.13. The *range* of a partial function $f : X \to \mathcal{L}(Y)$ is the predicate

$$\mathsf{range}_f : Y \to \mathsf{Prop}$$

defined by

$$\mathsf{range}_f(y) \stackrel{\mathsf{def}}{=} \exists(x : X), f(x) = \eta y.$$

The *domain* of f is the predicate

$$\mathsf{dom}_f : X \to \mathsf{Prop}$$

defined by

$$\mathsf{dom}_f(x) \stackrel{\mathsf{def}}{=} \mathsf{defined}(f(x)).$$

That is, y is in the range of f if there exists $x : X$ such that the value of $f(x)$ is y, and x is in the domain of f if f is defined at x.

5.4 Liftings are DCPOs

Given a type Y, we can define a type family $- \leq - : \mathcal{L}\, Y \to \mathcal{L}\, Y \to \mathcal{U}$ by

$$u \leq v \stackrel{\mathsf{def}}{=} \Sigma(t : \mathsf{defined}(u) \to \mathsf{defined}(v)), \Pi(p : \mathsf{defined}(u)), \mathsf{value}(u)(p) = \mathsf{value}(v)(t(q)).$$

For a general type Y, there may be multiple witnesses that $u \leq v$, but if Y is a set, then $- \leq -$ is proposition-valued. Moreover, we have

Theorem 5.14. *If Y is a set, then $(\mathcal{L}\, Y, \leq)$ is a directed-complete partial order with bottom $\bot$.*

Here, a directed-complete partial order X must have a prop-valued order relation, and for any inhabited family $u : I \to X$ such that $\Pi(i, j : I), \exists(k : I), u_i, u_j \leq k$, there is a least upper bound in X.

Proof. Let Y be a set. We need to see that $\leq$ is proposition-valued, so let $x, y : u \leq v$. Then we have

$$(x = y) \simeq \Sigma(p : \mathsf{defined}(u) \to \mathsf{defined}(v)), \Pi(p : \mathsf{defined}(u)), \mathsf{value}(u)(p) = \mathsf{value}(v)(t(q)).$$

As Y is a set, we know that $\mathsf{value}(u)(p) = \mathsf{value}(v)(t(q))$ is a proposition, and as $\mathsf{defined}(u)$ is a proposition, we have that $\Pi(p : \mathsf{defined}(u)), \mathsf{value}(u)(p) = \mathsf{value}(v)(t(q))$ is as well. Then $x = y$ is equivalent to a sum of propositions over a proposition.

If $u \leq v$ and $v \leq w$, then $u \leq w$ by composition in the first coordinate and concatenation in the second. We have that $u \leq u$ by $(\mathsf{id}, \lambda p.\mathsf{refl})$. Finally, let $u \leq v$ and $v \leq u$. Then we have $t : \mathsf{defined}(u) \to \mathsf{defined}(v)$ and $t' : \mathsf{defined}(v) \to \mathsf{defined}(u)$, and by proposition extensionality we have $p : \mathsf{defined}(u) = \mathsf{defined}(v)$. We wish to see that $u = v$, so we need to see that $\mathsf{transport}^{\lambda P.P \to Y}(p, \mathsf{value}(u)) = \mathsf{value}(v)$. This is equivalent to $\mathsf{value}(u) \circ \mathsf{idtofun}(p^{-1}) = \mathsf{value}(v)$. This in turn is equivalent to

$$\Pi(q : \mathsf{defined}(v)), \mathsf{value}(u)(t'(q)) = \mathsf{value}(v)(q),$$

and this type has an element by the assumption that $v \leq u$. In other words, we have an element $p : \mathsf{defined}(u) = \mathsf{defined}(v)$ and $\mathsf{value}(u) =_p \mathsf{value}(v)$. So then $u = v$.

Given a directed family $u_i : I \to \mathcal{L}Y$, take the extent of definition of the join u_∞ to be $\mathsf{defined}(u_\infty) \overset{\mathsf{def}}{=} \|\Sigma(i : I), \mathsf{defined}(u_i)\|$. Then by construction we have $\mathsf{isProp}(\mathsf{defined}(u_\infty))$. To define the value, we first define a function $\varphi : \big(\Sigma(i : I), \mathsf{defined}(u_i)\big) \to Y$ by

$$\varphi(i, p) \overset{\mathsf{def}}{=} \mathsf{value}(u_i)(p).$$

As the family is directed, for any i and j, there is a k such that $u_i, u_j \leq u_k$. So then we have for any $p : \mathsf{defined}(u_i)$ and $q : \mathsf{defined}(u_j)$ we have

$$\varphi(i, p) = \varphi_i(p) = \varphi_k(p') = \varphi_k(q') = \varphi_j(q) = \varphi(j, q)$$

where $\varphi_i = \mathsf{value}(u_i)$, for $p', q' : \mathsf{defined}(u_k)$ witnessing that $u_i \leq u_k$ and $u_j \leq u_k$ respectively. That is, φ is constant. So we have by Theorem 3.9 that φ factors through $\mathsf{defined}(u_\infty)$ as $\varphi_\infty \circ |-|$. So take $\mathsf{defined}(u_\infty) \overset{\mathsf{def}}{=} \varphi_\infty$.

We have $\mathsf{defined}(u_i) \to \mathsf{defined}(u_\infty)$ by $p \to |(i, p)|$. If $v : \mathcal{L}Y$ with $\Pi(i : I), u_i \leq v$, we have a map $(\Sigma(i : I), u_i) \to \mathsf{defined}(v)$, and this factors through u_∞ as $\mathsf{defined}(v)$ is a proposition. $\square$

5.5 Dominances and partial functions

The traditional way to examine restricted classes of partial functions [75, 44, 89] is to restrict the available extents of definition via *dominances*—subsets of Prop satisfying certain closure properties. We develop this approach here.

Definition 5.15. A *set of propositions* is a map $\mathsf{d} : \mathcal{U} \to \mathcal{U}$ together with

D1 a map $\Pi(X : \mathcal{U}), \mathsf{isProp}(\mathsf{d}(X))$,

D2 a map $\Pi(X : \mathcal{U}), \mathsf{d}(X) \to \mathsf{isProp}(X)$.

Note that these pieces of data are propositions. We say that d is *proposition valued* when it satisfies D1, and that d *selects propositions* when it satisfies D2.

Given any set of propositions d, the type $\mathcal{L}_{\mathsf{d}}(Y)$ of d-*partial elements* of a type Y is given by

$$\mathcal{L}_{\mathsf{d}}(Y) \overset{\mathsf{def}}{=} \Sigma(P : \mathcal{U}), \mathsf{d}(P) \times (P \to Y).$$

The next lemma is simple but invaluable, and we will make tacit use of it.

Lemma 5.16. *Any set of propositions* $\mathsf{d} : \mathcal{U} \to \mathcal{U}$ *is closed under equivalence. That is, if* $\mathsf{d}(P)$ *and* $P \simeq Q$*, then* $\mathsf{d}(Q)$*.*

Proof. As d selects propositions, we have $\mathsf{isProp}(P)$. Then since $Q \simeq P$, we know $\mathsf{isProp}(Q)$, and so $P = Q$ by proposition extensionality. Hence, $\mathsf{d}(Q)$, by transport. □

We may then define the d-partial functions $X \rightharpoonup_{\mathsf{d}} Y$ to be functions

$$X \rightharpoonup_{\mathsf{d}} Y \stackrel{\mathsf{def}}{=} X \to \mathcal{L}_{\mathsf{d}} Y.$$

In general, the d-partial functions might not be well-behaved: we do not necessarily have that the d-partial functions compose, or that total functions can be viewed as d-partial. In order to ensure these two properties, Rosolini introduced the notion of dominance.

Definition 5.17. A *dominance* is a set of propositions $\mathsf{d} : \mathcal{U} \to \mathcal{U}$, which we call *dominant*, satisfying

D3 The unit type is dominant: we have $\upsilon : \mathsf{d}(1)$,

D4 d has *conditional conjunction*: For any $P, Q : \mathcal{U}$ we have

$$\mathsf{d}(P) \to (P \to \mathsf{d}(Q)) \to \mathsf{d}(P \times Q).$$

Note that both pieces of data are property by D1.

In [6], Bauer calls D4 *the dominance axiom*, but does include D3 in the definition of dominance; since he works in the internal language of a topos, D1 and D2 are redundant. Reus and Streicher [68] call D4 *dependent conjunction*; since they are axiomatizing a particular dominance (the *Rosolini dominance* of Section 5.7), the relevant versions of D1, D2, and D3 are theorems.

There are three trivial examples of dominances:

$$\mathsf{d}_1(X) \stackrel{\mathsf{def}}{=} \mathsf{isContr}(X), \quad \mathsf{d}_2(X) \stackrel{\mathsf{def}}{=} (X = 0) + (X = 1), \quad \mathsf{d}_\Omega \stackrel{\mathsf{def}}{=} \mathsf{isProp},$$

with liftings corresponding to

$$\mathcal{L}_{\mathsf{d}_1}(X) \stackrel{\mathsf{def}}{=} X, \quad \mathcal{L}_{\mathsf{d}_2}(X) \stackrel{\mathsf{def}}{=} X + 1, \quad \mathcal{L}_{\mathsf{d}_\Omega}(X) \stackrel{\mathsf{def}}{=} \mathcal{L}(X),$$

In a type theoretic context, it is usually more convenient to let the conditional proposition Q in D4 also depend on the witness of P. Generalizing D4 to this case gives us

D4′ we have map

$$\sigma : \Pi(P:\mathcal{U}), (Q : P \to \mathcal{U}), \mathsf{d}(P) \to (\Pi(p:P), \mathsf{d}(Q(p))) \to \mathsf{d}(\Sigma(p:P), Q(p)).$$

This condition is perhaps more immediately recognizable: it says that a set of propositions is closed under Σ. In fact, we have this already from conditional conjunction.

Lemma 5.18. *For a set* d *of propositions,* D4 *is equivalent to* D4′*.*

Proof. For fixed P and Q, the constant family $\lambda(p:P).Q(p)$ is such that $P \times Q = \Sigma(p:P), Q(p)$ and $P \to Q = \Pi(p:P), Q(p)$, so closure under Σ gives D4.

Conversely, let d satisfy D4 and let $P:\mathcal{U}$ and $R : P \to \mathcal{U}$ such that $\mathsf{d}(P)$ and $\Pi(p:P), \mathsf{d}(R(p))$. Define

$$Q \stackrel{\mathsf{def}}{=} \Sigma(p:P), R(p).$$

We need to see that $\mathsf{d}(Q)$. First note that for any $p:P$ we have $R(p) = Q$, by $r \mapsto (p, r)$ with inverse $(p', r') \mapsto \mathsf{transport}(-, r')$, where "$-$" is the path $p' = p$ which exists since P is a proposition. But then we have $\Pi(p:P), d(Q)$, by assumption that R is d-valued. By the condition, we then have $\mathsf{d}(P \times Q)$. But $Q \simeq P \times Q$ by $(p, r) \mapsto (p, (p, r))$ with inverse given again by transport. □

The following result is the motivation for the notion of dominance.

Theorem 5.19. *A set of propositions* d *is a dominance iff* $\mathcal{L}_{\mathsf{d}} : \mathcal{U} \to \mathcal{U}_1$ *is a submonad of* $\mathcal{L} : \mathcal{U} \to \mathcal{U}_1$.

Proof. Let d be a dominance. The map $\mathcal{L}_{\mathsf{d}}\, Y \to \mathcal{L}\, Y$ is given by inclusion. We only need to see that the unit and Kleisli extension for $\mathcal{L}$ respect $\mathcal{L}_{\mathsf{d}}$. As $\mathsf{d}(1)$, we have that $\eta(y)$ is a d-proposition. We need to see that for all $(P, -, \varphi) : \mathcal{L}_{\mathsf{d}}(X)$ and $f : X \to \mathcal{L}_{\mathsf{d}}(Y)$, that the extent of $f^\sharp(P, -\varphi)$ is d-partial. But the extent of $f^\sharp(P, -, \varphi)$ is

$$\Sigma(p:P), \mathsf{defined}(f(\varphi(p))),$$

but by assumption P is a d-proposition and $\mathsf{defined}(f(\varphi(p)))$ is d-partial, so by condition D4′, we are done.

Now let $\mathcal{L}_{\mathsf{d}}$ be a submonad of $\mathcal{L}$, and let $\rho : \mathsf{d}(P)$ and $\sigma : P \to \mathsf{d}(Q)$. Define

$$\begin{aligned} p &: 1 \to \mathcal{L}_{\mathsf{d}}(P) \\ p(\star) &\stackrel{\mathsf{def}}{=} (P, \rho, \mathsf{id}) \end{aligned}$$

and

$$\begin{aligned} q &: P \to \mathcal{L}_{\mathsf{d}}(1) \\ q(x) &\stackrel{\mathsf{def}}{=} (Q, \sigma(x), \lambda w.\star) \end{aligned}$$

As $\mathcal{L}_{\mathsf{d}}$ is a monad, we have

$$q^\sharp \circ p : 1 \to \mathcal{L}_{\mathsf{d}}(1).$$

Moreover, $\mathcal{L}_{\mathsf{d}}$ is a submonad of $\mathcal{L}$, so we can calculate,

$$q^\sharp(p(\star)) = q^\sharp(P, -, \mathsf{id}) = ((\Sigma(p : P), Q), w, v).$$

We do not need to calculate the explicit value of w , as we have that w inhabits a proposition. We also do not need to calculate the explicit value of v, since we know there is a unique function $X \to 1$ for any type X. $\square$

As with the dominance of all propositions, we can consider relations valued in a dominance. That is, a d-*valued* relation between X and Y is a relation $R : X \to Y \to \mathcal{U}$ such that

$$\Pi(x : X), \mathsf{d}(\Sigma(y : Y), R(x, y)).$$

Theorem 5.20. *A set of propositions* d *is a dominance iff the* d*-valued relations are closed under composition and contain the identity relation.*

Proof. It is easy to check that the identity relation is d-valued iff $\mathsf{d}(1)$ since $\Sigma(y : Y), x = y$ is contractible.

Let d be a dominance and consider d-valued relations $R : X \to Y \to \mathcal{U}$ and $S : Y \to Z \to \mathcal{U}$. We need to see that for any $x : X$, the type

$$\Sigma(z : Z), \big(R; S(x, z)\big) \overset{\mathsf{def}}{=} \Sigma(z : Z), (y : Y), R(x, y) \times R(y, z)$$

is dominant. By the reshuffling map $(z, (y, (r, s))) \mapsto ((y, r), (z, s))$, this type is equivalent to

$$\sum_{(y,r):\Sigma(y:Y),R(x,y)} \Sigma(z : Z), S(y, z).$$

By assumption, both $\Sigma(y : Y), R(x, y)$ and $\Sigma(z : Z), S(y, z)$ are dominant. As d is closed under Σ, we have that $R; S$ is d-valued.

Conversely, let the d-valued relations be closed under composition, and suppose $\mathsf{d}(P)$ and $P \to \mathsf{d}(Q)$. Define $R : 1 \to P \to \mathcal{U}$ and $S : P \to 1 \to \mathcal{U}$ by

$$R(\star, p) \overset{\mathsf{def}}{=} P,$$

and

$$S(p, \star) \overset{\mathsf{def}}{=} Q.$$

For any $p : P$ we have $\Sigma(u : 1), S(p, u) \simeq S(p, \star) = Q$, so that

$$\Pi(p : P), \mathsf{d}(\Sigma(u : 1), S(p, u)),$$

That is, S is d-valued. Similarly, we have $(\Sigma(p : P), P) \simeq (P \times P) \simeq P$, so that R is d-valued. Then by assumption, $R; S$ is d-valued. Calculating $R; S(\star, \star)$ we see

$$R; S(\star, \star) \overset{\mathsf{def}}{=} \Sigma(p : P), P \times Q.$$

As P is a proposition, this type is equivalent to $P \times Q$. □

Collecting Theorems 5.19 and 5.20 and recalling Definition 3.18, we have

Theorem 5.21. *The following are equivalent for any set of propositions* d*:*

1. d *is a dominance;*
2. $\mathcal{L}_{\mathsf{d}}$ *is a submonad of* $\mathcal{L}$.
3. *There is a composition function*

$$- \square - : (Y \to \mathcal{L}_{\mathsf{d}}(Z)) \to (X \to \mathcal{L}_{\mathsf{d}}(Y)) \to (X \to \mathcal{L}_{\mathsf{d}}(Z)),$$

   restricting Kleisli composition for the $\mathcal{L}$ *monad.*
4. *The* d*-valued relations are closed under composition.*

Theorems 5.19 and 5.20 express essentially the same fact: the d-partial functions are closed under composition iff d is a dominance. Theorem 5.19 expresses this fact by viewing d-partial functions $X \rightharpoonup_{\mathsf{d}} Y$ as functions $X \to \mathcal{L}_{\mathsf{d}}(Y)$, while theorem 5.20 does so by viewing d-partial as d-valued relations. We can relate the two views by restricting the functions ele and rel given above for the dominance of all propositions.

Theorem 5.22. *Assuming univalence, for any set of propositions* d*, the maps* ele *and* rel *lift to a section-retraction pair on the* d*-valued relations and* d*-partial functions.*

Proof. Letting $X \rightharpoonup^{\mathsf{R}}_{\mathsf{d}} Y$ be the type of d-valued relations from X to Y, the lifted functions are

$$\begin{aligned} r &: (X \rightharpoonup^{\mathsf{R}}_{\mathsf{d}} Y) \to (X \to \mathcal{L}_{\mathsf{d}}(Y)) \\ r((R,\varphi)) &= \lambda x.\Big(\Sigma(y:Y), R(x,y)\ ,\ \varphi, \mathsf{pr}_0\Big), \end{aligned}$$

and

$$\begin{aligned} s &: (X \to \mathcal{L}_{\mathsf{d}}(Y)) \to (X \rightharpoonup^{\mathsf{R}}_{\mathsf{d}} Y) \\ s(f) &= \lambda x, y.(\Sigma(p : f_e(x)), f_v(p) = y, w), \end{aligned}$$

where w arises from the fact that $f_e(x)$ is always a d-proposition, and that

$$\big(\Sigma(y:Y), \Sigma(p : f_e(x)), f_v(p) = y\big) \simeq f_e(x),$$

using Lemma 5.16. The identity $r \circ s \sim \mathsf{id}$ follows directly from proposition extensionality. $\square$

The pair of functions given above are the restriction of the maps in Lemma 5.5. They fail to determine an equivalence, because we require equality between general types. Hence, they would be equivalences in the presence of univalence.

Corollary 5.23. *Assuming univalence, for any set of propositions* d *and* $X, Y : \mathcal{U}$*, the type* $X \to \mathcal{L}_{\mathsf{d}}(Y)$ *is equivalent to the type*

$$\Sigma(R : X \to Y \to \mathcal{U}), \Pi(x : X), \mathsf{d}(\Sigma(y : Y), R(x, y)).$$

Lemma 5.5 also restricts to an arbitrary dominance. As it relies on the fact that when Y is a set, the sum $\Sigma(y : Y), A(y)$ is a proposition iff each $A(y)$ is (Lemma 3.5), this may be surprising: the corresponding fact does not hold for an arbitrary dominance. However, it is not the dominance of propositions that gives us Lemma 3.5, but the h-level of propositions.

Corollary 5.24. *If* Y *is a set, then for any set of propositions* d *and any type* $X : \mathcal{U}$*, the type* $X \to \mathcal{L}_{\mathsf{d}}(Y)$ *is equivalent to the type*

$$\Sigma(R : X \to Y \to \mathcal{U}), \Pi(x : X), \mathsf{d}(\Sigma(y : Y), R(x, y)).$$

5.6 Size issues

Because the lifting raises universe levels, $\mathcal{L}$ and $\mathcal{L}_{\mathsf{d}}$ do not form monads as defined in Section 3.5. However, if we restrict our attention to propositions at universe $\mathcal{U}_0$, then the universe levels do not go past $\mathcal{U}_1$: the lifting operation is a monad on $\mathcal{U}_1$. We use the current section to explain this.

If $X : \mathcal{U}_0$, then $\mathcal{L}(X) : \mathcal{U}_1$, so $\mathcal{L}$ raises universe levels. However, it turns out that the raising stops at $\mathcal{U}_1$, assuming we take Prop to be Prop_0. Our operations are of the form $\mathsf{d} : \mathcal{U}_0 \to \mathcal{U}_0$, so for each j and $X : \mathcal{U}_j$, we have $\mathcal{L}^j$,

$$\mathcal{L}^j_{\mathsf{d}}(X) \stackrel{\mathsf{def}}{=} \Sigma(P : \mathcal{U}_0), \mathsf{d}(P) \times P \to X.$$

Since this quantification is over $\mathcal{U}_0$, we have that $\mathcal{L}^j_{\mathsf{d}}(X) : \mathcal{U}_{\mathsf{max}(j,1)}$. That is,

$$\mathcal{L}^j_{\mathsf{d}} : \mathcal{U}_j \to \mathcal{U}_{\mathsf{max}(j,1)}.$$

In other words, for $j > 0$, we have $\mathcal{L}_{\mathsf{d}}^j : \mathcal{U}_j \to \mathcal{U}_j$. If we use resizing axioms, then we would have $\mathsf{Prop} : \mathcal{U}_0$, so that the quantification in the definition of $\mathcal{L}(X)$ can be over a type in $\mathcal{U}_0$. Then $\mathcal{L}_d^0 : \mathcal{U}_0 \to \mathcal{U}_0$, and lifting would not raise universe levels at all.

When d satisfies D3, the unit η can similarly be indexed by universe levels, with type

$$\eta^j : \Pi(X : \mathcal{U}_j), X \to \mathcal{L}^j(X).$$

Similarly, when d satisfies D4′, the Kleisli operator has type

$$\Pi(X : \mathcal{U}_j), (Y : \mathcal{U}_k), (X \to \mathcal{L}^k(Y)) \to \mathcal{L}^j(X) \to \mathcal{L}^k(Y),$$

so Kleisli composition has type

$$\Pi(X : \mathcal{U}_j), (Y : \mathcal{U}_k), (Z : \mathcal{U}_l), (Y \to \mathcal{L}^l(Z)) \to (X \to \mathcal{L}^k(Y)) \to (X \to \mathcal{L}^l(Z)).$$

The point here is that even though lifting raises universe levels, it does so in a coherent way. In particular, if $X, Y, Z : \mathcal{U}_0$, then we have that Kleisli composition has type

$$(Y \to \mathcal{L}^0(Z)) \to (X \to \mathcal{L}^0(Y)) \to (X \to \mathcal{L}^0(Z)).$$

It is worth noting, moreover that the set of propositions we are most interested in, the *Rosolini propositions* (Definition 5.25) arises in such a way that Rosolini liftings are equivalent to a small type (Lemma 5.27), so the lifting of interest is predicative.

5.7 Rosolini propositions

A particularly important dominance in synthetic domain theory is the dominance of *Rosolini propositions*. We approach Rosolini propositions via the extended naturals, $\mathbb{N}_\infty$. Recall the map $\langle - \rangle : \mathbb{N}_\infty \to \mathcal{U}$ which takes a sequence (with at most one 1) to the proposition that it takes the value 1 (Section 4.1). The *Rosolini propositions* are the types in the image of this map. Explicitly,

Definition 5.25. $P : \mathcal{U}$ is Rosolini when

$$\|\Sigma(u : \mathbb{N}_\infty), P = \langle u \rangle\| .$$

Traditionally, the family of Rosolini propositions is denoted with Σ, which we avoid here due to notational clashes; we will use isRosolini.

Alternatively we may define a *Rosolini structure* over P to be a $u : \mathbb{N}_\infty$ such that $P = \langle u \rangle$. That is,

$$\mathsf{rosoliniStructure}(P) \stackrel{\mathsf{def}}{=} \Sigma(u : \mathbb{N}_\infty), P = \langle u \rangle,$$

so that $\mathsf{isRosolini}(P) = \|\mathsf{rosoliniStructure}(P)\|$.

The Rosolini propositions arise from computational considerations: if P is a Rosolini proposition, we can see a sequence $\alpha : \mathbb{N}_\infty$ such that $P \Leftrightarrow \langle \alpha \rangle$ as a "semi-decision procedure" for P. Then, a proposition is Rosolini if there exists a semi-decision procedure for it. However, this procedure is abstract: we have made no assumption that there is an actual algorithm for such a procedure. In Chapter 8, we will consider semi-decidable propositions and examine to what extent the Rosolini and semi-decidable propositions align.

The Rosolini propositions also have a connection to analysis, discovered together with Auke Booij: call a proposition P *Cauchy* if there exists a Cauchy real number $r : \mathbb{R}$ such that

$$P \Leftrightarrow (0 < r).$$

That is,

$$\mathsf{isCauchy}(P) \stackrel{\mathsf{def}}{=} \|\Sigma(r : \mathbb{R}), P \Leftrightarrow (0 < r)\| .$$

Here the Cauchy reals are defined following Bishop [9] as equivalence classes of regular sequences $s : \mathbb{N} \to \mathbb{Q}$. A sequence $s : \mathbb{N} \to \mathbb{Q}$ is *regular* when,

$$\mathsf{isRegular}(s) \stackrel{\mathsf{def}}{=} \Pi(n, m : \mathbb{N}), |s_n - s_m| \leq \frac{1}{m+1} + \frac{1}{n+1}.$$

We will not discuss here the full construction of $\mathbb{R}$; what matters for us is that for any real number $r : \mathbb{R}$, there exists a sequence of rationals converging to it:

$$\Pi(r : \mathbb{R}), \exists(s : \mathbb{N} \to \mathbb{Q}), \mathsf{isRegular}(s) \times \left(\Pi(n : \mathbb{N}), |r - s_n| < \frac{1}{n+1}\right).$$

Theorem 5.26. *A proposition is Cauchy iff it is Rosolini.*

Proof. Let P be Rosolini. As being Cauchy is a proposition, we can untruncate the witness that P is Rosolini to get $\alpha : \mathbb{N}_\infty$ such that $P = \langle \alpha \rangle$. Define $s : \mathbb{N} \to \mathbb{Q}$ by

$$s_n = \sum_{m \leq n} \frac{\alpha_n}{n}.$$

Then s is regular. We have that the real $r : \mathbb{R}$ defined by s is greater than 0 iff $\exists(n : \mathbb{N}), s_n > 0$, which happens precisely if $\exists(n : \mathbb{N}), \alpha_n = 1$. Then we have

$$P \Leftrightarrow \langle \alpha \rangle \Leftrightarrow (0 < r).$$

Conversely, let P be Cauchy. As being Rosolini is a proposition, we have $r : \mathbb{R}$ such that $P = (0 < r)$, and hence

$$\exists(s : \mathbb{N} \to \mathbb{Q}), \mathsf{isRegular}(s) \times \left(\Pi(n : \mathbb{N}), |r - s_n| < \frac{1}{n+1}\right),$$

and again, since we are trying to prove a proposition, we may assume an explicit regular sequence s.

Define $\beta : 2^{\mathbb{N}}$ by $\beta_n \stackrel{\mathsf{def}}{=} (0 \leq s_n + 1/n)$. Then take $\alpha : \mathbb{N}_\infty$ to be the truncation of β. We have

$$\langle \alpha \rangle \Leftrightarrow \left(\Sigma(n : \mathbb{N}), (0 \leq s_n + \frac{1}{n}\right) \Leftrightarrow (0 \leq r) \Leftrightarrow P. \qquad \square$$

It is worth noting that countable choice is not used in the above argument.

Before we consider whether the Rosolini propositions form a dominance, let us examine the relationship to the delay monad and its quotient by weak bisimilarity. Observe that we may form a lifting relative to Rosolini structures by

$$\mathcal{L}_{\mathsf{RS}}(X) \stackrel{\mathsf{def}}{=} \Sigma(P : \mathcal{U}), \mathsf{rosoliniStructure}(P) \times (P \to X).$$

Note that since $\mathsf{rosoliniStructure}$ is not a proposition, the same element $x : X$ has multiple representatives in $\mathcal{L}_{\mathsf{RS}}(X)$. Nevertheless we can form a (non-canonical) inclusion $\eta : X \to \mathcal{L}_{\mathsf{RS}}(X)$ by taking $\eta(x)$ to be $(\langle\alpha\rangle, (\alpha, \mathsf{refl}), \lambda u.x)$ where $\alpha = \overline{0} = \lambda n.n = 0$. This makes $\mathcal{L}_{\mathsf{RS}}$ a reorganization of the definition of the delay monad.

Lemma 5.27. *For any type X, there is an equivalence*

$$\mathcal{L}_{\mathsf{RS}}(X) \simeq \mathsf{D}(X).$$

Moreover, $\eta : X \to \mathcal{L}_{\mathsf{RS}}(X)$ lifts over this equivalence to $\mathsf{now} : X \to \mathsf{D}(X)$.

Proof. Simply manipulate the definition of $\mathcal{L}_{\mathsf{RS}}(X)$. We have

$$\begin{aligned}
\mathcal{L}_{\mathsf{RS}}(X) &= \Sigma(P : \mathcal{U}), \mathsf{rosoliniStructure}(P) \times (P \to X) \\
&= \Sigma(P : \mathcal{U}), \Sigma(\mu : \mathbb{N}_\infty), (P = \langle\mu\rangle) \times (P \to X) \\
&\simeq \Sigma(\mu : \mathbb{N}_\infty), \langle\mu\rangle \to X \\
&= \Sigma(\mu : \mathbb{N}_\infty), (\Sigma(n : \mathbb{N}), \mu_n = 1) \to X \\
&\simeq \Sigma(\mu : \mathbb{N}_\infty), \Pi(n : \mathbb{N}), X^{\mu = \overline{n}} \\
&\simeq \mathsf{D}(X).
\end{aligned}$$

□

We have a map $q : \mathcal{L}_{\mathsf{RS}}(X) \to \mathcal{L}_{\mathsf{isRosolini}}(X)$ given by

$$q(P, d, \varphi) \stackrel{\mathsf{def}}{=} (P, |d|, \varphi).$$

Theorem 5.28. *The map $q : \mathcal{L}_{\mathsf{RS}}(X) \to \mathcal{L}_{\mathsf{isRosolini}}(X)$ is surjective. Moreover, when X is a set, there is an equivalence*

$$\mathcal{L}_{\mathsf{isRosolini}}(X) \simeq D(X)/\approx$$

commuting with the quotient map $D(X) \to D(X)/\approx$.

$$\begin{tikzcd}
\mathcal{L}_{\mathsf{RS}}(X) \arrow[r, "\simeq"] \arrow[d, "q"] & \mathsf{D}(X) \arrow[d, "q"] \\
\mathcal{L}_{\mathsf{isRosolini}}(X) \arrow[r, "\simeq"] & \mathsf{D}(X)/\approx
\end{tikzcd}$$

Proof. To show that q is surjective, we need to see that for any $x : \mathcal{L}_{\mathsf{isRosolini}}(X)$ the fiber of q over x is inhabited. I.e., we want an inhabitant of

$$\|\Sigma(y : \mathcal{L}_{\mathsf{RS}}(X)), q(y) = x\|.$$

Consider $x = (P, r, \varphi)$, so that we have $r : \|\mathsf{RS}(X)\|$. As we are trying to show a proposition, we may assume we have some $r' : \mathsf{RS}(X)$. Then for $y = (P, r', \varphi)$, we have $q(y) = x$.

Composing q with the equivalence $e : \mathsf{D}(X) \to \mathcal{L}_{\mathsf{RS}}(X)$ determines a map

$$f : \mathsf{D}(X) \to \mathcal{L}_{\mathsf{isRosolini}}(X).$$

It is clear that f respects bisimilarity; if $x, y : \mathsf{D}(X)$ are bisimilar, then we must have that the sequences determining $\mu_x, \mu_y : \mathbb{N}_\infty$ defining x and y are such that

$$\langle \mu_x \rangle \Leftrightarrow \langle \mu_y \rangle;$$

when x and y do take a value, it must be the same value, and so $\mathsf{value}(f(x)) = \mathsf{value}(f(y))$.

Then f factors through a map

$$f/\approx : \mathsf{D}(X)/\approx \to \mathcal{L}_{\mathsf{isRosolini}}(X),$$

so long as $\mathcal{L}_{\mathsf{isRosolini}}(X)$ is a set; this happens whenever X is a set. This map is again surjective, as f is.

It remains to show that the extension $f/\approx$ has propositional fibers. It is enough to show that if $f(x) = f(y)$ then $x \approx y$. We have that $f(x) = f(y)$ precisely when both $\mu_x = \mu_y$ and if $\mu_x(n) = \mu_y(n) = 1$, then x and y are equal to $\mathsf{delay}^n(a)$ for some $a : X$. These two conditions are the definition of bisimilarity. $\square$

5.8 Choice principles and the dominance axiom

The Rosolini propositions form a dominance in, for example, the effective topos [75], but in general we cannot show that they do. We give in this section a characterization of the amount of choice needed to show that the Rosolini propositions form a dominance, starting with a weakening of countable choice.

Theorem 5.29. *For any type A, the following are equivalent*

1. *Choice from $\mathbb{N}$ to families of the form $n \mapsto (\alpha_n = 1) \to A$, with $\alpha : 2^{\mathbb{N}}$.*
2. *Choice from Rosolini propositions to A.*

Proof. Consider the following propositions:

(1) $\langle \alpha \rangle \to \|A\|$,

(2) $(\Sigma(n : \mathbb{N}), \alpha_n = 1) \to \|A\|$,

(3) $\Pi(n : \mathbb{N}), ((\alpha_n = 1) \to \|A\|)$,

(4) $\Pi(n : \mathbb{N}), \|(\alpha_n = 1) \to A\|$,

(5) $\|\Pi(n : \mathbb{N}), (\alpha_n = 1) \to A\|$,

(6) $\|(\Sigma(n : \mathbb{N}), \alpha_n = 1) \to A\|$,

(7) $\|\langle \alpha \rangle \to A\|$.

The implication $(4) \to (5)$ is the above instance of countable choice, and the implication $(1) \to (7)$ is the above instance of propositional choice. Note that $(3) \to (4)$ holds because $\alpha_n = 1$ is decidable. Hence the chain of implications $(1) \to (2) \to (3) \to (4) \to (5) \to (6) \to (7)$ gives propositional choice from countable choice, and the chain of implications $(4) \to (3) \to (2) \to (1) \to (7) \to (6) \to (5)$ gives countable choice from propositional choice. □

The above form of countable choice, where A takes the form $\mathsf{rosoliniStructure}(P)$ for some P is sufficient to prove that the Rosolini propositions form a dominance. In fact, we can do no better.

Theorem 5.30. *The following are equivalent*

1. *Choice from Rosolini propositions to Rosolini structures. I.e.,*

$$\mathsf{isRosolini}(P) \to (P \to \|\mathsf{rosoliniStructure}\, Q\|) \to \|P \to \mathsf{rosoliniStructure}(Q)\|$$

2. *The Rosolini propositions form a dominance.*

We omit the proof as we generalize this theorem to Theorem 5.32 below.

Corollary 5.31. *Choice from $\mathbb{N}$ to families of the form $n \mapsto (\alpha_n = 1) \to A$ implies that the Rosolini propositions form a dominance. Hence, countable choice implies that the Rosolini propositions form a dominance.*

Proof. We need to see that choice from $\mathbb{N}$ to families of the form $n \mapsto (\alpha_n = 1) \to A$ gives us Rosolini choice. So let P and Q be propositions. We wish to see

$$\|\mathsf{rosoliniStructure}(P)\| \to (P \to \|\mathsf{rosoliniStructure}(Q)\|) \to \|P \to \mathsf{rosoliniStructure}(Q)\| .$$

In fact, we may untruncate $\mathsf{rosoliniStructure}(P)$, so let $\alpha : \mathbb{N}_\infty$ such that $P = \langle\alpha\rangle$. Then we need

$$(\langle\alpha\rangle \to \|\mathsf{rosoliniStructure}(Q)\|) \to \|\langle\alpha\rangle \to \mathsf{rosoliniStructure}(Q)\| .$$

Note that for all n, we have $(\alpha_n = 1) \to \langle\alpha\rangle$. So then setting

$$A \stackrel{\mathsf{def}}{=} \mathsf{rosoliniStructure}(Q),$$

we have $(\alpha_n = 1) \to \|A\|$. Since $\alpha_n = 1$ is decidable, we have choice from $\alpha_n = 1$ (Theorem 3.23). Hence, we have

$$\Pi(n : \mathbb{N}), \|(\alpha_n = 1) \to A\| .$$

This is exactly the assumption of our choice principle, so we conclude

$$\|\Pi(n : \mathbb{N}), (\alpha_n = 1) \to A\| .$$

As $(\Pi(n : \mathbb{N}), (\alpha_n = 1) \to A) \simeq (\langle\alpha\rangle \to A)$, we have

$$\|\langle\alpha\rangle \to A\| .$$

And this concluses the proof, since

$$\|\langle\alpha\rangle \to A\| \simeq \|P \to \mathsf{rosoliniStructure}(Q)\| . \qquad \square$$

The reliance on choice we saw in Theorem 5.30 is an instance of a more general phenomenon.

Theorem 5.32. *Let* $\mathsf{D} : \mathcal{U} \to \mathcal{U}$ *select propositions* (D2) *and have conditional conjunction* (D4) *and define* $\mathsf{d}(X) \stackrel{\mathsf{def}}{=} \|\mathsf{D}(X)\|$*. Then the following are equivalent*

1. *Choice from* d*-propositions to* D*-structures; i.e., for all* $X, Y : \mathcal{U}$*,*

$$\mathsf{D}(X) \to (X \to \|\mathsf{D}(Y)\|) \to \|X \to \mathsf{D}(Y)\|$$

2. *The* d*-propositions satisfy the dominance axiom.*

Proof. We need to show that under the assumptions of $\|\mathsf{D}(X)\|$ and $X \to \|\mathsf{D}(Y)\|$ that we have

$$\|X \to \mathsf{D}(Y)\| \simeq \|\mathsf{D}(X \times Y)\| .$$

By properties of truncation, it is enough to assume $\mathsf{D}(X)$ and $X \to \|\mathsf{D}(Y)\|$ and give maps

$$(X \to \mathsf{D}(Y)) \leftrightarrow \mathsf{D}(X \times Y).$$

From left to right is simply the fact that D has conditional conjunction. From right to left, if $(X \to \mathsf{D}(Y))$ then $X \to \mathsf{isProp}(Y)$. Since $X \to ((X \times Y) = Y)$ when X and Y are propositions, we have $(X \to \mathsf{D}(Y)) \to (X \to \mathsf{D}(X \times Y))$. By modus ponens, we are done. $\square$

Rosolini choice (choice from Rosolini propositions to Rosolini structures) is actually quite weak. We saw above that it follows from even a weakening of countable choice, but it is quite a bit weaker than countable choice.

Theorem 5.33. *Each of the following principles alone implies Rosolini choice:*

1. *countable choice;*
2. *the weakening of countable choice in Theorem 5.29;*
3. *"untruncated" LPO:* $\Pi(\alpha : 2^{\mathbb{N}}), \langle\alpha\rangle + (\alpha = \lambda n.0)$*;*

4. *"untruncated" WLPO:* $\Pi(\alpha : 2^{\mathbb{N}}), \langle\alpha\rangle + \neg\langle\alpha\rangle$;
5. *"truncated" WLPO:* $\Pi(\alpha : 2^{\mathbb{N}}), \|\langle\alpha\rangle + \neg\langle\alpha\rangle\|$;
6. *Propositional choice: for all* $P, A : \mathcal{U}$ *with* $\mathsf{isSet}(A)$ *and* $\mathsf{isProp}(P)$,

$$(P \to \|A\|) \to \|P \to A\| .$$

Proof. 1. This follows from 2;

2. Rosolini choice is the specialization of this principle to types of the form $\mathsf{rosoliniStructure}(Q)$ for some $Q : \mathcal{U}$;
3. LPO implies WLPO, so this follows from 4;
4. This follows from 5, since the truncated form is weaker;
5. Let $\mathsf{isRosolini}\ P$ and let Q be a proposition such that $P \to \mathsf{isRosolini}(Q)$. Then

$$\exists(\alpha : 2^{\mathbb{N}}), P \simeq \langle\alpha\rangle.$$

   By WLPO, we then have $\|P + \neg P\|$. However, P is a proposition, so then so is $P + \neg P$, and we have $P + \neg P \simeq \|P + \neg P\|$. Then we may perform a case analysis on P: if P is true, then so is $\|\mathsf{rosoliniStructure}(Q)\|$, and so we have $\|P \to \mathsf{rosoliniStructure}(Q)\|$. If P is false, then we vacuously have $\|P \to \mathsf{rosoliniStructure}(Q)\|$.
6. If $\mathsf{isRosolini}(P)$, then P is a proposition, so by propositional choice we have

$$(P \to \mathsf{isRosolini}(Q)) \to \|P \to \mathsf{rosoliniStructure}(Q)\| . \qquad \square$$

In light of the above, it is worth wondering how far we can go towards specifying a class of partial maps via a class of propositions—indeed we cannot prove that any dominance is different from both isProp and $\mathsf{isContr}$ without violating classicality.

So, if we were to work with dominances, choice would be unavoidable. The first attempt at a solution is to allow $\mathsf{D}(X)$ to be structure, rather than property. This commits us to handling structure explicitly; in particular, we would need to use definitions and constructions which respect structure. This seems unduly cumbersome, but we develop this line of reasoning somewhat in the next section. We will subsequently make use of this development in finding an approach to partial functions which works for our purposes.

Note that Rosolini choice is not enough to give a version of Theorem 5.14 for Rosolini propositions. However, full countable choice is enough to show that the Rosolini propositions are closed under *countable* joins.

Theorem 5.34. *If countable choice holds, then the Rosolini propositions are closed under countable joins in the lattice of all propositions. Hence, countable choice implies that the Rosolini lifting is an ω-CPO.*

Proof. Let $P_i : \mathbb{N} \to \mathsf{Prop}$ be a countable chain of Rosolini propositions. We need to see that $\|\Sigma(i : \mathbb{N}), P_i\|$ is a Rosolini proposition. We have by assumption,

$$\Pi(i : \mathbb{N}), \|\Sigma(\alpha_i : \mathbb{N}_\infty), P_i = \langle \alpha_i \rangle\| \, .$$

By countable choice we then have

$$\|\Pi(i : \mathbb{N}), \Sigma(\alpha_i : \mathbb{N}_\infty), P_i = \langle \alpha_i \rangle\| \, .$$

Since being Rosolini is a proposition, to prove that $\|\Sigma(i : \mathbb{N}), P_i\|$ is Rosolini, it is enough to prove this from

$$\Pi(i : \mathbb{N}), \Sigma(\alpha_i : \mathbb{N}_\infty), P_i = \langle \alpha_i \rangle.$$

Define $\beta : \mathbb{N}_\infty$ to be the sequence such that $\beta(n) = 1$ iff there is a lexicographically minimal $(i, j) : \mathbb{N} \times \mathbb{N}$ with $i + j = n$ such that $\alpha_i(j) = 1$. It is clear that $\mathsf{isProp}\langle \beta \rangle$.

Now observe that $\|\Sigma(i : \mathbb{N}), P_i\|$ is equivalent to $\|\Sigma(i, j : \mathbb{N}), \alpha_i(j) = 1\|$. By construction we have that

$$\langle \beta \rangle \simeq \|\Sigma(i, j : \mathbb{N}), \alpha_i(j) = 1\| \, .$$

Then β provides a witness that $\|\Sigma(i : \mathbb{N}), P_i\|$ is Rosolini. $\square$

5.9 Structural Dominances

In order to define the notion of disciplined map below in Section 5.10, we need to relax the notion of dominance to allow $\mathsf{D}(X)$ to be structured. This structural version of a dominance is as follows.

Definition 5.35. A *structural dominance* is a map $\mathsf{d} : \mathcal{U} \to \mathcal{U}$ together with witnesses of Conditions D2, D3, and D4′. That is, we have

D2. a map $\Pi(X : \mathcal{U}), \mathsf{d}(X) \to \mathsf{isProp}(X)$,

D3. The unit type is dominant: we have $\upsilon : \mathsf{d}(1)$,

D4.′ we have map

$$\sigma : \Pi(P : \mathcal{U}), (Q : P \to \mathcal{U}), \mathsf{d}(P) \to (\Pi(p : P), \mathsf{d}(Q(p))) \to \mathsf{d}(\Sigma(p : P), Q(p)).$$

Note that D3 and D4′ are structure rather than property in the absence of D1.

Lemma 5.18 gives an equivalence between D4 and D4′ under the assumption that d satisfies both D1 and D2. It is enough to show only logical equivalence because both D4 and D4′ are propositions in this case. When we only have D2, this is not enough. It seems that in this case, we can only show that D4′ is a retract of D4.

Before giving the argument in detail, a sketch is useful: let d select propositions, and fix $P : \mathcal{U}$ with $\mathsf{d}(P)$. The type of closure under conditional conjunction, when applied to a type Q is the same as the type of Σ-closure applied to $\lambda p.Q$, since we have defined $P \times Q$ as $\Sigma(p : P), Q$ and $P \to \mathsf{d}(Q)$ as $\Pi(p : P), \mathsf{d}(Q)$. Conversely, since P is a proposition, we can show that any type family $Y : P \to \mathcal{U}$ is equal to the constant type family $\lambda q.\Sigma(p : P), Y(p)$. Then given a witness of Σ-closure, we get a witness of closure under conditional conjunction, by treating a type as a constant type family; given a witness of closure under conditional conjunction, we get a witness of closure under Σ by passing back and forth along this equality.

We will give the required homotopy by applying the following lemma.

Lemma 5.36. *Let X be a type and $A, B : X \to \mathcal{U}$ with $f : \Pi(x : X), A(x) \to B(x)$. Let $p : x = x'$ for $x, x' : X$. Then*

$$\mathsf{idtofun}(\mathsf{ap}_B(p^{-1})) \circ f_x \circ \mathsf{idtofun}(\mathsf{ap}_A(p)) = f_{x'}.$$

Proof. By path induction on p. In the case that $p = \mathsf{refl}$, both $\mathsf{ap}_A(p)$ and $\mathsf{ap}_B(p)$ are reflexivity, and so we need to see $f_x = f_x$, which is obvious. $\square$

For fixed $P : \mathcal{U}$ and $d : \mathsf{d}(P)$, define the type families $A, B : (P \to \mathcal{U}) \to \mathcal{U}$ by

$$A(Y) = \Pi(p : P), \mathsf{d}(Y p)$$

and

$$B(Y) = \mathsf{d}(\Sigma(p : P), Y p).$$

Note that for $Q : \mathcal{U}$, we have that $P \to \mathsf{d}(Q) = A(\lambda p.Q)$ and $\mathsf{d}(P \times Q) = B(\lambda p.Q)$. So we can define

$$\begin{aligned} E : \big(\Pi(Y : P \to \mathcal{U}), A(Y) \to B(Y)\big) &\to \Pi(Q : \mathcal{U}), A(\lambda p.Q) \to B(\lambda p.Q) \\ E(\theta) &\overset{\mathsf{def}}{=} \lambda Q.\theta(\lambda p.Q) \end{aligned}$$

Defining the candidate retraction requires some more work.

Lemma 5.37. *For $X : \mathcal{U}$ and $Y : X \to \mathcal{U}$, let $Z \overset{\mathsf{def}}{=} \Sigma(x : X), Y(x)$. If $\mathsf{isProp}(X)$, then*

$$\Pi(x : X), (Y(x) \simeq Z),$$

and

$$Z \simeq (X \times Z).$$

Proof. For the first equivalence, fix $x : X$, we have $Y(x) \to Z$ by $y \mapsto (x, y)$, with inverse given by transport.

Moreover, we have $Z \to X \times Z$ by $(x, y) \mapsto (x, x, y)$ with inverse again given by transport. $\square$

In the case where Y is also proposition valued (that is, $\Pi(x : X), \mathsf{isProp}(Yx)$), these equivalences are equalities, as all types involved are propositions. Moreover, the above gives us an equality of predicates, by function extensionality.

Lemma 5.38. *If X is a proposition and $Y : X \to \mathcal{U}$ is a predicate, then*

$$Y = \lambda x.\Sigma(x : X), Y(x).$$

This lemma allows us to define, for our fixed $P : \mathcal{U}$.

$$F : \big(\Pi(Q : \mathcal{U}), A(\lambda p.Q) \to B(\lambda p.Q)\big) \to \Pi(Y : P \to \mathcal{U}), A(Y) \to B(Y)$$
$$F(\delta) \stackrel{\mathsf{def}}{=} \lambda Y.\, \mathsf{idtofun}(\mathsf{ap}_B(w_Y^{-1})) \circ \delta_{\Sigma(p:P),Yp} \circ \mathsf{idtofun}(\mathsf{ap}_A(w_Y))$$

where $w_Y : Y \to \lambda p.\Sigma(p : P), Y(p)$ arises from Lemma 5.38. Finally, we can compute the composite $F \circ E$:

Lemma 5.39. *The map E is a section of the map F.*

Proof. Fix $\theta : \Pi(Y : P \to \mathcal{U}), A(Y) \to B(Y)$. Then

$$F(E(\theta)) = \lambda Y.\, \mathsf{idtofun}(\mathsf{ap}_B(w_Y^{-1})) \circ \theta_{\lambda p.\Sigma(p:P),Yp} \circ \mathsf{idtofun}(\mathsf{ap}_A(w_Y))$$

Note that θ has the type of f in Lemma 5.36, with $X = P \to \mathcal{U}$, and so we have for each Y,

$$F(E(\theta))(Y) = \theta(Y),$$

and so $F \circ E$ is homotopic to the identity. ☐

In the above, we fixed $P : \mathcal{U}$ and $d : \mathsf{d}(P)$. Letting P and d vary again, we see that closure under Σ is a retract of closure under conditional conjunction That is, we have

Corollary 5.40. *If $\mathsf{d} : \mathcal{U} \to \mathcal{U}$ selects propositions, then the type that d is closed under Σ is a retract of the type*

$$\Pi(P, Q : \mathcal{U}), \mathsf{d}(P) \to (P \to \mathsf{d}(Q)) \to \mathsf{d}(P \times Q).$$

The main example of a structural dominance we are interested in is the dominance of Rosolini structures.

Lemma 5.41. *The map* $\mathsf{rosoliniStructure}$ *is a structural dominance.*

Proof. D2 is immediate. For υ we take $\overline{0}$. To get σ, by Lemma 5.40 and the definition of $\mathsf{rosoliniStructure}$, we first define a conditional addition $- +^* - : \Pi(\alpha : \mathbb{N}_\infty), (\langle\alpha\rangle \to \mathbb{N}_\infty) \to \mathbb{N}_\infty$, such that

$$\langle \alpha +^* \beta \rangle \Leftrightarrow \exists(k : \mathbb{N}), (\alpha_k = 1) \times \exists(j : \mathbb{N}), \beta(k)_j = 1.$$

In fact, this specification is almost a complete definition. Define,

$$(\alpha +^* \beta)_n = 1 \Leftrightarrow \exists(k \leq n), (\alpha_k = 1) \times \exists(j \leq n), \beta(k)_j = 1.$$

Since this is a bounded quantification of decidable predicates, this is again a decidable property on $\mathbb{N}$. Moreover, since there can be at most one such k and j, we have $\mathsf{isProp}(\langle \alpha +^* \beta \rangle)$.

Now, we define σ as follows. Fix $P : \mathcal{U}$ and $Q : P \to \mathcal{U}$ with $\alpha : \mathbb{N}_\infty$ with $w : \langle\alpha\rangle = P$ and for each $p : P$ a $\beta_p : \mathbb{N}_\infty$ such that $Q(p) = \langle \beta_p \rangle$. We then have a map $\beta' : \langle\alpha\rangle \to \mathbb{N}_\infty$ with $\beta'(a) = \beta_p$ where a transports over w to p. Then a Rosolini structure for $\Sigma(p : P), Q(p)$ is given by $\alpha +^* \beta'$ together with the fact that $\alpha +^* \beta'$ takes value 1 precisely when there is $a : \langle\alpha\rangle$, and $\beta'(a)$ takes value 1. $\square$

5.10 Disciplined maps

The takeaway from Section 5.8 above is that the Rosolini partial functions are not useful in the absence of choice principles: we need choice a principle to compose Rosolini partial functions. Another approach is necessary. We could instead work with Rosolini structures—with the delay monad—but doing this means carrying around extra information, in particular, distinguishing between partial elements based not on their eventual value, but on how long it takes to compute this value. This is unsatisfactory; instead, we need a predicate on partial functions which is closed under composition without choice.

The predicate on partial functions given by a dominance arises by restricting the available extents of definition. Instead, we will give a predicate on partial functions directly: Any proposition-index family $\mathsf{D} : \mathsf{Prop} \to \mathcal{U}$ gives rise to a function from the lifting relative to that family $\mathcal{L}_{\mathsf{D}}(Y)$ to the general lifting $\mathcal{L}(Y)$, and so we get a map from D-partial functions $X \to \mathcal{L}_{\mathsf{D}}(Y)$ to general partial functions $X \to \mathcal{L}(Y)$. If D is a set of propositions, this map is an embedding, and so $X \to \mathcal{L}_{\mathsf{D}}(Y)$ is equivalent to its image, but this is not true if D is not proposition-valued. Then, instead of looking at $X \to \mathcal{L}_{\mathsf{D}}(Y)$, we will look at its image in $X \to \mathcal{L}(Y)$.

We can think of the space $X \to \mathcal{L}_{\mathsf{D}}(Y)$ as being too wild for our purposes, so we first *tame* the space; the tamed maps we will call *disciplined*. The rest of the section lays this out precisely.

Given any family $\mathsf{D} : \mathcal{U} \to \mathcal{U}$ which selects propositions and $X : \mathcal{U}$, we have a map

$$\begin{aligned} e_{\mathsf{D}} &: \mathcal{L}_{\mathsf{D}}(Y) \to \mathcal{L}(Y) \\ e_{\mathsf{D}}(P, d, \varphi) &= (P, -, \varphi), \end{aligned}$$

where the omitted "$-$" follows from Condition D2. By post-composition, this gives us a map

$$\mathsf{tame} : (X \to \mathcal{L}_{\mathsf{D}}(Y)) \to (X \to \mathcal{L}(Y)).$$

This map always factors through $(X \to \mathcal{L}_{\mathsf{d}}(Y))$, where $\mathsf{d}(Y) = \|\mathsf{D}(Y)\|$, and if D satisfies Condition D1, then this map is an embedding. We will thus call e_D the *canonical embedding*, even when D carries structure.

We call a function in the image of $\mathsf{tame}_{\mathsf{D}}$ D-*disciplined*. That is,

Definition 5.42. A function $f : X \to \mathcal{L}(Y)$ is D-*disciplined* if it is in the image of $\mathsf{tame}_{\mathsf{D}}$. Formally,

$$\begin{aligned} \mathsf{isDis}_{\mathsf{D}} &: (X \to \mathcal{L}(Y)) \to \mathcal{U} \\ \mathsf{isDis}_{\mathsf{D}}(f) &\overset{\text{def}}{=} \exists (f' : X \to \mathcal{L}_{\mathsf{D}}(Y)), \mathsf{tame}(f') = f \\ \mathsf{Dis}_{\mathsf{D}}(X, Y) &: \mathcal{U} \\ \mathsf{Dis}_{\mathsf{D}}(X, Y) &\overset{\text{def}}{=} \Sigma(f : X \to \mathcal{L}(Y)), \mathsf{isDis}_{\mathsf{D}}(f) \end{aligned}$$

Lemma 5.43. *If* D *is a structural dominance, for any* $f : X \to \mathcal{L}\,Y$ *and* $g : Y \to \mathcal{L}\,Z$*, we have*

$$\mathsf{isDis}_{\mathsf{D}}(f) \to \mathsf{isDis}_{\mathsf{D}}(g) \to \mathsf{isDis}_{\mathsf{D}}(g \,\square\, f).$$

Proof. As $\mathsf{isDis}(g \mathbin{\square} f)$ is a proposition, we may assume $f' : X \to \mathcal{L}_{\mathsf{D}}(Y)$ and $g' : Y \to \mathcal{L}_{\mathsf{D}}(Z)$ such that $\mathsf{tame}(f') = f$ and $\mathsf{tame}(g') = g$. We need $h' : X \to \mathcal{L}_{\mathsf{D}}(Z)$ with $\mathsf{tame}(h') = g \mathbin{\square} f$. We claim that $h' \stackrel{\mathsf{def}}{=} g' \mathbin{\square} f'$ suffices. Note that tame is post-composition with e, that e acts independently on the components of $z : \mathcal{L}_{\mathsf{D}}(Z)$, and that e is the identity on extent and value, so to see that $e(g'^{\sharp}(f'x)) = g^{\sharp}(fx)$, we need only the component witnessing that $\mathsf{defined}(x)$ satisfies D. However, since this component takes values in a proposition, there is nothing to check. □

Theorem 5.44. *If* D *is a structural dominance, then there is a composition operator,*

$$- \diamond - : \mathsf{Dis}_{\mathsf{D}}(Y, Z) \to \mathsf{Dis}_{\mathsf{D}}(X, Y) \to \mathsf{Dis}_{\mathsf{D}}(X, Z).$$

Proof. Immediate from Lemma 5.43. □

Observe that since the truncation d of any structural dominance is proposition-valued, the disciplined maps are a subset of the d-partial functions. Also notice that if D is in fact a dominance, then for any $f : X \to \mathcal{L}\,Y$ the type

$$\Sigma(g : X \to \mathcal{L}_{\mathsf{D}}(Y)), \mathsf{tame}(f) = g$$

is a proposition, and so we can remove the truncation. That is, for a dominance D, we have that the D-disciplined maps are exactly the D-partial functions. The situation is more interesting when D is not proposition-valued. In this case, a lifting

$$\mathsf{Dis}_{\mathsf{D}}(X, Y) \to (X \to \mathcal{L}_{\mathsf{D}}(Y))$$

corresponds to untruncating the existential in the definition of isDis. To show this rigorously, we use the following lemma.

Lemma 5.45. *For any* $f : X \to \mathcal{L}(Y)$ *and any structural dominance* D*, the function*

$$F_f : \big(\Pi(x : X), \mathsf{D}(\mathsf{defined}(fx))\big) \to \Sigma(g : X \to \mathcal{L}_{\mathsf{D}}(Y)), f = \mathsf{tame}(g)$$

defined by

$$F_f(d) = \big(\lambda x.(f_e(x), d(x), f_v(x))\ \ w\big)$$

where w is the equality arising from the tuple $(\mathsf{refl}, -, \mathsf{refl})$*, is an equivalence.*

Proof. Letting

$$A(f) \stackrel{\mathsf{def}}{=} \Pi(x : X), \mathsf{D}(\mathsf{defined}(f(x))),$$

and

$$B(f) \stackrel{\mathsf{def}}{=} \Sigma(g : X \to \mathcal{L}_{\mathsf{D}}(Y)), f = \mathsf{tame}(g),$$

we have equivalences

$$\begin{aligned}\Sigma(f : X \to \mathcal{L}(Y)), A(f) &\simeq X \to \mathcal{L}_{\mathsf{D}}(Y) \\ &\simeq \Sigma(f : X \to \mathcal{L}(Y)), B(f).\end{aligned}$$

The composite equivalence E is the identity on the first component, and F_f on the second. That is, E arises as F_Σ from Section 1.9, and so by Theorem 1.28, each F_f is an equivalence. □

Theorem 5.46. *Fix a structural dominance* D *and its associated set of propositions* d*. If countable choice holds, then for any $f : \mathbb{N} \to \mathcal{L}(\mathbb{N})$, we have that f is disciplined iff for all $n : N$, the extent of $f(n)$ is a* d*-proposition. Then, in particular, countable choice implies that $f : \mathbb{N} \to \mathcal{L}(\mathbb{N})$ is Rosolini-disciplined iff f factors through a Rosolini partial function.*

Proof. By the above lemma, we have an equivalence

$$\mathsf{isDis}(f) \simeq \|\Pi(n : \mathbb{N}), \mathsf{D}(\mathsf{defined}(fn))\|\,.$$

We know there is an implication

$$\|\Pi(n : \mathbb{N}), \mathsf{D}(\mathsf{defined}(fn))\| \to \Pi(n : \mathbb{N}), \mathsf{d}(\mathsf{defined}(fn)),$$

and the implication in the other direction is an instance of countable choice.

Finally, every disciplined map factors through a d-partial function. Moreover, we have that $f : \mathbb{N} \to \mathcal{L}(\mathbb{N})$ factors through a d-partial function iff $\Pi(n : \mathbb{N}), \mathsf{d}(\mathsf{defined}(fn))$, and by the above argument, this implies that f is disciplined. $\square$

Note that we cannot use the dominance choice principle for Theorem 5.46, since there does not seem to be a way to write the type $\Pi(n : \mathbb{N}), \|\mathsf{D}(\mathsf{defined}(fn))\|$ in the form $\Pi(a : A), \|\mathsf{D}(Pa)\|$, where A is dominant.

Note, moreover, that the above is a theorem about disciplined maps from $\mathbb{N}$. Since we will look at first-order computability theory, we are interested in partial functions from $\mathbb{N}$ to $\mathbb{N}$. In particular, the Rosolini-disciplined maps form the most likely candidate for a notion of partial function that can be consistently posited to be the computable functions.

5.11 Comparison with synthetic domain theory

In synthetic domain theory (for example, in [75, 44, 68, 83, 89]), a dominance is defined to be a map $\mathsf{d} : \Omega \to \Omega$ with the object of *all* partial elements of Y is given by

$$\{A \in \mathcal{P}Y \mid \forall x, y. A(x) \to A(y) \to x = y\},$$

and the object of d-partial elements of Y by

$$\{A \in \mathcal{P}Y \mid (\forall x, y. A(x) \to A(y) \to x = y) \land \mathsf{d}(\exists x. A(x))\}$$

which translates into type theory as

$$\Sigma(A : Y \to \mathsf{Prop}), (\Pi(x, y : Y), A(x) \to A(y) \to x = y) \times \mathsf{d}(\Sigma(x : Y), A(x)).$$

However, this does not work in a univalent setting, unless Y is a set. In particular, recall that all maps from the homotopy circle S^1 into Prop are constant (Lemma 2.68). In particular, we do not have the unit map $\mathsf{S}^1 \to \mathcal{L}\,\mathsf{S}^1$; indeed,

$$\eta(\mathsf{base}) = \lambda x.(x = \mathsf{base}),$$

is not valued in propositions. So we must change A to have type $Y \to \mathcal{U}$, but then we run into the issue that $A(x) \to A(y) \to x = y$ may have non-trivial structure, because the type $x = y$ is not a proposition. Again, we have that $(x = \mathsf{base}) \to (y = \mathsf{base}) \to x = y$ has numerous witnesses. In short, we must change the total lifting in two ways:

$$\underbrace{\Sigma(A : Y \to \mathsf{Prop})}_{\Sigma(A:X \to \mathcal{U})}, \underbrace{(\Pi(x, y : Y), A(x) \to A(y) \to x = y)}_{\mathsf{isProp}(\Sigma(x:Y), A(x))}.$$

This gives the lifting of Y as single-valued relations with 1. As we saw in Theorem 5.6 this is equivalent to the lifting we give in Definition 5.3, assuming univalence. Even without univalence, our lifting is a retract of this lifting. The application of univalence here occurs because we cannot establish equality between maps $Y \to \mathcal{U}$ without an extensionality principle for types. As a result, our lifting $\Sigma(p : \mathcal{U}), \mathsf{isProp}(P) \times (P \to X)$ is more useful in the absence of univalence: we need only proposition and function extensionality to establish equality between elements of $\mathcal{L}(Y)$.

Typical axiomatizations of synthetic domain theory include a subset of propositions, written Σ, which are our Rosolini propositions. In the settings of interest to synthetic domain theorists, the dominance axiom typically holds—usually as a consequence of countable choice. The axiomatization given by Reus [69] is of note for not requiring Σ to be a dominance. There, they are not interested in Σ to define liftings, but instead are interested in developing domain theoretic ideas abstractly. They give the following axiomatization of $\Sigma \subseteq \mathsf{Prop}$:

(aSig) True and false are in Σ;

(bSig) Σ is closed under $\vee$ and $\wedge$;

(cSig) Σ is closed under existential quantification over $\mathbb{N}$;

(dSig) Σ embeds into Prop;

(PHOA) Σ^Σ is equivalent to the graph of the standard ordering relation on $\Sigma \times \Sigma$;

(S) For any $P : (\mathbb{N} \to \Sigma) \to \Sigma$ such that $P(\lambda n.\top)$, there exists an $n : \mathbb{N}$ such that $P(\overline{n})$, where $\overline{n} = \lambda k.k < n$;

(MP) propositions in Σ have double-negation elimination.

Axioms (aSig), (bSig), and (dSig) are true for the Rosolini propositions, while (cSig) and (S) require some amount of choice. In fact, (cSig) implies Rosolini choice in our context, so implies that the Rosolini propositions form a dominance. Note, however, that our proof relies on the fact that the Rosolini propositions arise via truncation from a structural dominance, in contrast to their direct axiomatization. Both (PHOA) and (MP) are not available in our setting: (PHOA) implies that there are propositions which are not Rosolini propositions (the negation of Kripke's schema, and hence negating excluded middle), and (MP) already implies we are working in a computation setting.

5.12 Discussion

The Rosolini dominance plays a fundamental role in Rosolini's PhD Thesis [75], which is concerned with investigating notions of effectiveness. It seems that the notion of effectiveness leads one inexorably to consider partiality, since the notion of dominance was introduced by him here specifically to deal with partiality in toposes, where (like in type theory), the notion of (total) function is primitive. Historically, the set of Rosolini propositions has been denoted Σ, clashing with type-theoretic notation. The Rosolini propositions correspond to the Sierpinski space in formal topology and to the Σ_1 propositions from the arithmetic hierarchy. Several traditional taboos can be stated in terms of Rosolini propositions:

- Kripke's Schema says that all propositions are Rosolini.
- Markov's Principle is double-negation elimination for Rosolini propositions.
- LPO says that true and false are the only Rosolini propositions.
- WLPO says that Rosolini propositions are either false or not false.

This view makes it clear that Kripke's Schema and Markov's Principle together imply excluded middle.

Until the last year or so, the relationship between partiality and dominances has only been studied in the context of synthetic domain theory and synthetic computability theory, where countable choice holds, and so the problem with composition is not apparent. The form of Choice in Theorem 5.29 was isolated by Martín Escardó when examining what was necessary to compose partial functions. Theorem 5.32 was generalized from Theorem 5.30, which we showed already in our paper [31]. In that paper, Theorem 5.28 was left as a conjecture, and Bas Spitters quickly proved it independently. The fact that the Rosolini-structure lifting is equivalent to the delay monad means that it does not raise universe levels. The same is true for the

Rosolini lifting. Besides the proof via Theorem 5.28, there is a result due to Egbert Rijke [71] giving conditions under which an operation does not increase universe levels.

The notion of disciplined map was proposed, but not worked out in the conclusion to [31], inspired partially by comments from Mike Shulman. Chapman, Uustalu, and Veltri [16] briefly proposed dealing with the problem of composition by viewing the quotiented delay monad as arrow (in the sense of [42]); this is morally the same as the approach via disciplined maps. To my knowledge, they have not pursued this line, although Uustalu and Veltri have continued to examine partiality and monads [87].

CHAPTER 6

Partiality in Bishop Mathematics

Our approach to partiality in Chapter 5 relies crucially on univalent definitions. However, it is worth examining the extent to which we can do the work contained there in a Bishop-style approach using setoids. Since the aim of this thesis is not to work with setoids, this chapter will be short and less detailed. Moreover, we will not use univalent definitions in this chapter. In particular, a proposition is simply any type, a relation is any type family, we do not use any h-levels, and we use the Curry-Howard interpretation of logic from Section 1.2.

We cannot approach general partial functions from the setoid perspective with our tools. First we require quantification over the universe, but there is no reasonable way to turn the universe into a setoid—no universe setoid was given by Martin Hofmann in his thesis on setoids [35], and it is not clear that is possible [23]. Second, we rely on the definition of Prop, which we are avoiding in the setoid approach. Instead, we will restrict our attention to functions arising from the *Rosolini lifting*, which can be treated as a setoid. Since the definition of Rosolini propositions we gave relies also on Prop, we will instead approach the Rosolini dominance in this chapter via the delay monad and its quotient by bisimilarity. Recall Lemma 5.27 and Theorem 5.28, which show that this presentations is sound. The setoid approach does resolve one difficulty from the previous chapter: countable choice holds in the setoid model. This result seems to be folklore, so we prove it in Section 6.6.

6.1 Setoids

Recall that a *setoid* is a type together with an equivalence relation. In this context, an *equivalence relation* on X is a type family $R : X \to X \to \mathcal{U}$ together with witnesses,

$$
\begin{aligned}
r &: \Pi(x : X), R(x, x) \\
(-)^{-1} &: \Pi(x, y : X), R(x, y) \to R(y, x) \\
(-) \cdot (-) &: \Pi(x, y, z : X), R(y, z) \to R(x, y) \to R(x, z).
\end{aligned}
$$

We will abuse notation and refer to a setoid by its underlying type. We will also use $\approx$ (or, when we need to be more precise, $\approx_X$) for the relation on a setoid X, while leaving the data for $\approx$ implicit.

The equivalence relation is meant to capture equality on the underlying type. This allows us to impose extensionality principles as we wish, at the cost of bookkeeping. For example, if we wish to have function extensionality, we equip each function type $X \to Y$ with the equivalence relation

$$f \approx g \stackrel{\mathsf{def}}{=} \Pi(x : X), f(x) \approx g(x).$$

The bookkeeping required is substantial: we must ensure that any construction on setoids respects equivalence. In particular, if X and Y are (the underlying types of) setoids, we will call an element of the type $X \to Y$ an *operation*, and we will say that an operation f is *extensional* or is *a function* when

$$\Pi(x, y : X), (x \approx y) \to f(x) \approx f(y).$$

Then the type of functions from X to Y is given by

$$\Sigma(f : X \to Y), \Pi(x, y : X), (x \approx y) \to f(x) \approx f(y).$$

Except when we state otherwise, we take $\approx$ on function types to be extensional equality:

$$(f, w) \approx (g, u) \overset{\mathsf{def}}{\Leftrightarrow} \Pi(x : X), f(x) \approx g(x).$$

Note that in this definition of $\approx$, we are treating the evidence that f is a function as property, regardless of the underlying type. We will consequently talk about a function $f : X \to Y$ rather than an operation $f : X \to Y$ with a witness $w : \Pi(x, y : X), (x \approx y) \to f(x) \approx f(y)$.

6.2 Families of setoids

The requirement that a function f respects equivalence corresponds to the function ap_f. In Part 1.7, we followed the definition of ap_f with a dependent version apd_f which relied on transport. As a consequence of the uniform definition of identity types, we were able to construct transport for general type families. Since the equivalence relation on a setoid is ad-hoc, we have to give our notion of transport in an ad-hoc way. The typical way to do this is via what Palmgren [64] calls *proof-irrelevant* families of setoids, which treat the equivalence relation on the index type as property. A more nuanced approach, called *proof-relevant* families by Palmgren, treats the equivalence relation on the index type as structure. The context is a setoid $(A, \approx)$ and a family of types $B : A \to \mathcal{U}$ where each $B(a)$ is a setoid. In order to be a family of setoids, we need at least *reindexing* bijections

$$\varphi : \Pi(x, y : A), x \approx y \to (B(x) \simeq B(y)).$$

We call B a *proof-irrelevant family* of setoids when

(i) $\varphi_p \approx \mathsf{id}_{B_x}$ whenever $p : x \approx x$ and
(ii) $\varphi_q \circ \varphi_p \approx \varphi_r$ whenever $p : x \approx y$, $q : y \approx z$ and $r : x \approx z$.

We call B a *proof-relevant family* of setoids when

(a) $\varphi_{r(x)} \approx \mathsf{id}_{B_x}$ for all $x : X$;
(b) $\varphi_{p^{-1}} \circ \varphi_p \approx \mathsf{id}_{B(x)}$ and $\varphi_p \circ \varphi_{p^{-1}} \approx \mathsf{id}_{B(y)}$ for any $p : x \approx y$; and
(c) $\varphi_q \circ \varphi_p \approx \varphi_{q \cdot p}$ whenever $p : x \approx y$ and $q : y \approx z$.

Note that any proof-irrelevant family of setoids is also a proof-relevant family of setoids. In either case, we then have a version of transport, or substitution given by the reindexing bijections. We can then say that a dependent function $f : \Pi(x : A), B(x)$ is *extensional* when B is a family of setoids and

$$\Pi(x, y : A), \Pi(p : x \approx y), \varphi_p(f(x)) \approx f(y).$$

In other words, a dependent function on setoids is extensional when it satisfies a version of apd.

We are interested also in predicates $P : A \to \mathcal{U}$. For these cases, we usually don't care about any setoid structure on $P(a)$, so let us say that a *extensional predicate* on a setoid A is a type family $P : A \to U$ with reindexing bijections

$$\mathsf{transport} : \Pi(x, y : A), x \approx y \to (P(x) \simeq P(y)).$$

6.3 Truncation and quotients

A setoid X is *proof-irrelevant* when $x \approx y$ for all $x, y : X$. This notion of proof-irrelevance is a setoid-version of being a homotopy proposition. Indeed, given a setoid X, we have the setoid $\|X\|$ with $x \approx_{\|X\|} y$ for all $x, y : X$. Then the operation $| - | : X \to \|X\|$ which is identity on elements is indeed a function. Moreover, $\|X\|$ has the universal property of truncation with respect to proof-irrelevant setoids: let P be any proof-irrelevant setoid, and $f : X \to P$ be a function. Then the operation f is also a function $\|X\| \to P$, as for any $x, y : X$ we have that $f(x) \approx f(y)$, and so in particular, when $x \approx_{\|X\|} y$, we have $f(x) \approx f(y)$. Moreover, if we take two functions $\|X\| \to P$ to be equal when they are extensionally equal as operations, then $f : \|X\| \to P$ is the unique factorization of $f : X \to P$ through $\|X\|$.

We can generalize this construction to any equivalence relation in the expected way. Let X be a setoid and let R be an equivalence relation which respects $\approx$. That is, R is a family of types such that $x \approx y \to R(x, y)$. Then the quotient X/R of X by R is the setoid with base type X and equivalence relation R. Any function $X \to Y$ which respects $\approx$ will then also respect R, and so extends (uniquely) to a function $X/R \to Y$.

Even more generally, Dybjer and Moeneclaey [27] showed that any Higher-inductive type whose highest constructor is a path between points (i.e., of type $x = y$ for some elements x, y) can be interpreted in the setoid model.

6.4 The Delay monad

We now turn to the delay monad. We need to ensure that the definition given in Chapter 4 makes sense in the context of setoids. To do this, we need to define $\mathbb{N}_\infty$. In turn, we need setoid representations of $\mathbb{N}$ and 2. Fortunately, this last task is easy: we equip $\mathbb{N}$ and 2 with the equality relation given by the identity type. Note that for $\mathbb{N}$ we can define this without having the identity type using Theorem 1.13. Equality on 2 can be represented in a similar way.

Then we may define $\mathbb{N}_\infty$ to be the setoid whose underlying type is

$$\Sigma(\alpha : 2^{\mathbb{N}}), \Pi(j, k : \mathbb{N}), (\alpha_j = 1) \to (\alpha_k = 1) \to (j = k),$$

and whose equivalence relation is extensional equality on the first component: $(\alpha, -) \approx (\beta, -)$ when $\Pi(n : \mathbb{N}), \alpha_n = \beta_n$. Then, for a setoid X, we may define $\mathsf{D}(X)$ via one of the representations in Chapter 4:

$$\mathsf{D}(X) \stackrel{\mathsf{def}}{=} \Sigma(\mu : \mathbb{N}_\infty), \Pi(n : \mathbb{N}), (\mu_n = 1) \to X,$$

The equivalence relation that gives rise to the setoid $\mathsf{D}(X)$ is

$$(\mu, \varphi) \approx (\nu, \psi) \stackrel{\mathsf{def}}{=} \Sigma(p : \mu \approx \nu), \Pi(n : \mathbb{N}), \mathsf{transport}(p, \varphi)(n) \approx_{(\nu_n = 1) \to X} \psi(n).$$

In order for this to be well-defined, we need to know that the type family

$$\Pi(n : \mathbb{N}), (\mu_n = 1) \to X$$

is a family of setoids.

In order to show that $\mathsf{D}(X)$ forms a monoid, we need to give η and $(-)^\sharp$, and moreover, show that these operations are extensional. Showing η to be extensional is easy. Showing $(-)^\sharp$ to be extensional is a little more work, but ultimately corresponds to the computation given in Theorem 4.9.

6.5 Rosolini partial elements and functions

By Theorem 5.28, we can represent the Rosolini lifting via the quotient of the delay monad by weak bisimilarity. To quotient by weak bisimilarity, we will replace the underlying type of $\mathsf{D}(X)$ with the (equivalent) uncurried form

$$\mathsf{D}(X) \overset{\mathsf{def}}{=} \Sigma(\mu : \mathbb{N}_\infty), \langle\mu\rangle \to X,$$

where $\langle\mu\rangle$ is again $\Sigma(n : \mathbb{N}), \mu_n = 1$, and consider instead the equivalence relation

$$(\mu, \varphi) \approx (\nu, \psi) \overset{\mathsf{def}}{=} (\langle\mu\rangle \leftrightarrow \langle\nu\rangle) \times \Big(\Pi(m : \langle\mu\rangle), \Pi(n : \langle\nu\rangle), \varphi(m) \approx \psi(n)\Big).$$

Note that by using the above representation instead of the curried form

$$\mathsf{D}(X) \overset{\mathsf{def}}{=} \Sigma(\mu : \mathbb{N}_\infty), \Pi(n : \mathbb{N}), (\mu_n = 1 \to X),$$

we avoid needing to know that $\Pi(n : \mathbb{N}), (\mu_n = 1) \to X$ is a family of setoids, since we do not need to reindex. Above, where we defined the equivalence relation on the unquotiented $\mathsf{D}(X)$, this reorganization doesn't help us, since we also needed the sequences to agree at every point.

6.6 The axiom of choice

It may come as a surprise that the monad data for the delay monad is also extensional with respect to this coarser equivalence relation; i.e., that the quotiented delay monad, $\mathsf{D}(-)/\approx$, is again a monad when we work with setoids. However, countable choice holds in the setoid model, on account of the fact that the setoid structure on $\mathbb{N}$ is discrete—that it arises from the identity type.

Theorem 6.1. *Let X be any setoid, and $R : \mathbb{N} \to X \to \mathcal{U}$ a relation such that for all $n : \mathbb{N}$ there exists an $x : X$ such that $R(n, x)$. Then there is a function $f : A \to B$ such that $R(n, f(n))$ for all $n : \mathbb{N}$.*

Proof. The hypothesis of the theorem gives us a witness

$$w : \Pi(n : \mathbb{N}), \Sigma(x : X), R(n, x).$$

Then we get an operation $f : \mathbb{N} \to X$ defined by

$$f(n) = \mathsf{pr}_0(w(n)),$$

and moreover, we have for any $n : \mathbb{N}$ that $\mathsf{pr}_0(w(n)) : R(n, f(n))$. All the remains is to see that f is extensional. Let $n, k : \mathbb{N}$ with $n \approx k$. But if $n \approx k$ we must have that $n = k$, and so $f(n) = f(k)$. $\square$

Unfortunately, retaining witness data is not enough for the full axiom of choice, since we need our operations to be extensional [58]. In more detail, the general situation is that we have a relation $R : A \to B \to \mathcal{U}$ such that for all $a : A$ there exists a $b : B$ with $R(a, b)$. We wish to find a choice function $f : A \to B$. The operation f can always be found in the setoid model, by projecting from the type $\exists(b : B), R(x, y)$, but this operation is not guaranteed to be extensional. For the setoid of natural numbers, the equivalence relation is such that all operations out of $\mathbb{N}$ are extensional, and so we have the above theorem.

6.7 Discussion

It is possible to construct the Rosolini lifting in the setoid approach, and moreover, this does form a monad, as a result of the fact that countable choice holds in the setoid model. Recent work by Coquand, Mannaa, and Ruch [22] gives a model of univalent type theory in which countable choice does not hold. To make the Rosolini propositions form a dominance, we could perhaps abandon the univalent setting in favor of setoids, but this approach won't allow us to place the Rosolini lifting in the context of a more general lifting. To resolve this, we could follow other authors (such as [35] and [21]), and introduce an impredicative type of propositions to have access to a type Prop that could be used to form a general lifting. In this case, to get the Rosolini lifting as a subtype of the lifting, we would need to ensure our axiomatization of Prop allowed us to isolate some of those propositions as the Rosolini propositions, and form a lifting setoid of those; whether the resulting lifting would be equivalent to the quotient of the delay monad, and whether this lifting would form a monad would depend on choices made about the behavior of Prop. Reus and Streicher [68] give one approach along similar lines, although they do not use setoids. It would be worth examining the interpretation of their type theory in the setoid model. Unless Prop is chosen to behave much like the univalent type of propositions, we lose the relationship between structure and property in this approach.

Nevertheless, the setoid approach is limited by the fact that there is no clear way to interpret the universe in setoids. Beyond the fact that this makes the univalent definition of propositions impossible, this strictly limits the types, the logical principles [81], and more importantly, the computable functions which can be described in our system. Indeed, work by Aczel [1], Feferman [32], Griffor and Rathjen [34], Rathjen [66, 67], and Setzer [78, 79, 80] shed light on the ordinal analysis of Martin-Löf type theory with universes. In short, adding universes to type theory allows the construction of larger ordinals; using a hierarchy of recursive functions, such as the Löb-Wainer hierarchy [52], these larger ordinals can be used to define faster growing computable functions. Consequently, having universes in our system gives a more fine-grained analysis of the limits of the partial computable functions. This thesis is focused on first-order computability, but we hope to set the stage for a constructive approach to studying computability at higher types, and computability with more interesting objects, such as metric spaces, manifolds and algebraic structures. In this case, we need a universe not only to define particular computable functions, but also to describe the objects we wish to compute with.

To summarize and add to the above, the setoid approach has the following advantages over the univalent approach:

1. The setoid approach allows us to use extensionality principles in a simpler version of MLTT [35]. In particular, quotients and truncations can be described without extending the type theory.
2. countable choice holds in the setoid model, so the Rosolini lifting becomes a monad without assuming additional principles. If we include a small type of propositions, we can indeed mimic the general lifting, and get the Rosolini lifting as a submonad of this.
3. The setoid approach fits squarely with the traditional account of constructive mathematics proposed by Bishop.

On the other hand, the univalent approach has the following advantages over the setoid approach

1. The univalent approach allows us to make use of universes; in particular, this allows us to define more computable functions, and to develop computability in more interesting domains.
2. Definitions are more uniform. As we use the inductively defined identity types instead of explicitly chosen equivalence relations, we can state much more general theorems, and all concepts are already extensional. Moreover, in the setoid approach, choice of representation is important: the equivalence relation on the quotient $\mathsf{D}(X)/{\approx}$ is more difficult to state for the representation

   $$\mathsf{D}(X) \stackrel{\mathsf{def}}{=} \Sigma(\mu : \mathbb{N}_\infty), \Pi(n : \mathbb{N}), (\mu_n = 1) \to X,$$

   as it requires us to make use of the fact that the type family $\Pi(n : \mathbb{N}), (\mu_n = 1) \to X$ is a family of setoids. This fact is not-obvious, and the result is somewhat technical. So with this representation, a non-trivial theorem is required to even form the quotient in the setoid approach.
3. The univalent perspective fits squarely with a structural account of mathematics, and extends the traditional account of constructive mathematics.
4. A type of propositions fits naturally into the theory, and clarifies the relationship between structure and property.
5. There are computer proof systems with native support for formalization in univalent styles. No such support exists for the setoid approach. In particular, cubicaltt [17] and Cubical Agda implement cubical type theory. There is some support for univalent mathematics in Agda, Coq, and Lean, but this support is given via axioms, so is less satisfactory.

Part III

Computability theory

We are now able to turn our attention to computability theory. In Chapter 7, we develop a theory of computability *as structure*, via a modest abstraction of Turing machines (which we call *recursive machines*), based on primitive recursive combinators. Here, we follow Section 1.6 in calling a type family $X \to \mathcal{U}$ a *structure* and calling a type family *property* when it is valued in propositions. In particular, in Chapter 7 we define a type family $\mathsf{CompStruct} : (\mathbb{N} \rightharpoonup \mathbb{N}) \to \mathcal{U}$ of *computation structures*, while in Chapter 8, we truncate this notion to arrive at the property $\mathsf{isComputable} : (\mathbb{N} \rightharpoonup \mathbb{N}) \to \mathsf{Prop}$.

Our abstraction is motivated by the fact that the initialization and transition functions of a Turing machine are primitive recursive, and so we can represent any Turing machine via a pair of (descriptions of) primitive recursive functions. This approach is easier to work with in our system than Turing machines or μ-recursive functions: Turing machines contain a great deal of data that needs to be tracked, while handling minimization takes more technical computation than handling the other basic recursive operations. We develop some basic results in computability theory using recursive machines, and there are no surprises here. The point is to give evidence that our notion of partial function is sound—that we can view the computable partial functions as certain functions $\mathbb{N} \to \mathcal{L}(\mathbb{N})$—and that recursive machines are indeed capable of being used for a development of computability theory. The only novelty from a computability theorist's perspective is the language. It is worth noting, on this regard that we have no need of specifying Kleene equality ("$f \approx g$ iff for all x, either both $f(x)$ and $g(x)$ are defined and equal, or both $f(x)$ and $g(x)$ are undefined"), since Kleene equality is simply equality in $\mathbb{N} \to \mathcal{L}\,\mathbb{N}$; our approach again allows us to dispense with an ad hoc notion of equality. This definition also allows us to remove the implicit use of excluded middle in the usual statement of Kleene equality.

In the first three sections of Chapter 8, we truncate notions of Chapter 7 to arrive at a notion of computability *as property*. Since most results have quite short proofs using Lemma 2.14, these sections are short. After this, we prove the undecidability of the halting problem, in particular demonstrating the existence of a function $d : \mathbb{N} \to \mathcal{L}(\mathbb{N})$ (Turing's diagonal function) which is not computable. Any total function we can define is computable, and moreover, constructive intuition tells us that a constructive definition should give rise to a computer program. Indeed, while it may be consistent in a constructive framework that all total functions $\mathbb{N} \to \mathbb{N}$ are computable, the partial function $d : \mathbb{N} \rightharpoonup \mathbb{N}$ is not computable. We return to the implications of this fact in Chapter 9, but first isolate a subset of the Rosolini propositions which we call *semidecidable*—those propositions for which there exists a computable witness that they are Rosolini. The semidecidable propositions are introduced and compared to computable functions in Section 8.5.

Chapter 9 turns to the question of which functions can be computable. We first (Section 9.1) demonstrate that computable partial functions and semidecidable maps fit into the Rosolini partial functions and Rosolini propositions, and then turn to the question of which functions are computable. As we have two notions of computability (structure and property), we can state Church's thesis in two ways: one in the logic of structures, and one in the logic of propositions. The former turns out to be false (Theorem 9.9), by an argument due to Troelstra [84]. The second is more plausible; the argument given by Troelstra can be reworked slightly to give a weaker result: Church's thesis is incompatible with the existence of an embedding from the computable functions into $\mathbb{N}$. This result can be leveraged to resolve the conflict between the topos-theoretic and type-theoretic facts discussed in the introduction. From there, we can consider a version of Church's thesis for partial functions. In fact, Church's thesis tells us that the Rosolini partial functions and the semidecidable partial functions coincide. Countable choice tells us that each type of partial functions coincides with the associated type of disciplined maps. Together, these results tell us (Theorem 9.14) that under Church's thesis and countable choice, the computable partial functions are the Rosolini partial functions. Since we know the effective topos to be a model of countable choice and Church's thesis, this suggests that it is consistent that all Rosolini partial functions are computable; unfortunately, a univalent version of the effective topos is needed to use this for a consistency result, and that requires techiques beyond what we consider here.

CHAPTER 7

Computability as structure

We develop in this chapter computability theory with computability as *structure* (C.f. 1.6) The results here are essentially the same as classical results in computability; as we shall see, the classical development of basic computability theory goes through with only minor changes. In particular,

- we use partial functions as in the previous chapter, as functions $X \to \mathcal{L}(Y)$;
- we make explicit reference to structure throughout.

This makes the statements of theorems slightly more cumbersome, but otherwise presents little change. In the next Chapter 8, we develop computability as property and compare it to computability as structure. For the most part, we get the standard results in computability theory for computability as property from the equivalent results for computability as structure, by functoriality of truncation. However, when we look at how computability interacts with the broader mathematical universe, things seem to change. For example, we have an embedding from "functions with computability structure" to $\mathbb{N}$, but such an embedding from "functions with the property of being computable" to $\mathbb{N}$ would imply a decidability result that we cannot expect to hold constructively.

7.1 Primitive recursion

In order to undertake a study of computability, we need to introduce a model of computation; Turing machines are intuitively convincing, but we would like a model that is somewhat less cumbersome to work with, so we will abstract away many internal details. We will make use of primitive recursive functions for this, so we briefly recall the definition and some basic facts. The material here can be found in any standard reference such as Odifreddi [62] or Rogers [74].

Definition 7.1. The type family $\mathsf{PR} : \mathbb{N} \to \mathcal{U}$ of *primitive recursive combinators (of arity n)* is defined inductively by

1. $\mathsf{s} : \mathsf{PR}_1$;
2. for any $n : \mathbb{N}$ and $k < n$ we have $\mathsf{p}^n_k : \mathsf{PR}_n$;
3. for any $n, k : \mathbb{N}$ we have $\mathsf{c}^n_k : \mathsf{PR}_n$;
4. if $f : \mathsf{PR}_n$ and $g_i : \mathsf{PR}_m$ for each $0 < i \leq n$, then $f\langle g_1, \ldots, g_n\rangle : \mathsf{PR}_m$.
5. if $f : \mathsf{PR}_{n+2}$ and $g : \mathsf{PR}_n$, then $\mathsf{r}_{f,g} : \mathsf{PR}_{n+1}$.

We will sometimes write PR for the total type $\Sigma(n : \mathbb{N}), \mathsf{PR}_n$. Similarly, we define a type family $\mathsf{PrimRec} : \Pi(n : \mathbb{N}), (\mathbb{N}^n \to \mathbb{N}) \to \mathcal{U}$ inductively by

1. $\mathsf{PrimRec}_1(\mathsf{succ})$;
2. for any $n : \mathbb{N}$ and $k < n$ we have $\mathsf{PrimRec}_n(\mathsf{pr}_k)$;
3. for any $n, k : \mathbb{N}$ we have $\mathsf{PrimRec}_n(\lambda x.k)$;
4. if $\mathsf{PrimRec}_n(f)$ and $\mathsf{PrimRec}_m(g_i)$ for each $0 < i \leq n$, then $\mathsf{PrimRec}_m(\lambda x.f(g_1(x), \ldots, g_n(x)))$
5. if $\mathsf{PrimRec}_{n+2}(f)$ and $\mathsf{PrimRec}_n(g)$, then $\mathsf{PrimRec}_{n+1}(\mathsf{rec}_{f,g})$, where

$$
\begin{aligned}
\mathsf{rec}_{f,g}(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\
\mathsf{rec}_{f,g}(k+1, x_1, \ldots, x_n) &= f(k, \mathsf{rec}_{f,g}(k, x_1, \ldots, x_n), x_1, \ldots, x_n),
\end{aligned}
$$

and where we leave the constructors for $\mathsf{PrimRec}_n(f)$ unnamed.

We can then define a function $\mathsf{app} : \Pi(n : \mathbb{N}), \mathsf{PR}_n \to \mathbb{N}^n \to \mathbb{N}$ in the obvious way:

$$
\begin{aligned}
\mathsf{app}(\mathsf{s}) &= \mathsf{succ}, \\
\mathsf{app}(\mathsf{p}^n_k) &= \lambda(x_1, \ldots, x_n).x_k, \\
\mathsf{app}(\mathsf{c}^n_k) &= \lambda x.k, \\
\mathsf{app}(f\langle g_1, \ldots, g_n\rangle) &= \lambda x.f(g_1(x), \ldots, g_n(x)), \\
\mathsf{app}(\mathsf{r}_{f,g}) &= \mathsf{rec}_{f,g}\,.
\end{aligned}
$$

It is easy to check that $\mathsf{PrimRec}(\mathsf{app}(t))$, and that app induces an equivalence

$$\mathsf{PR}_n \simeq \Sigma(f : \mathbb{N}^n \to \mathbb{N}), \mathsf{PrimRec}(f),$$

for all $n : \mathbb{N}$. We say that f has *primitive recursive structure* when $\mathsf{PrimRec}(f)$.

Definition 7.2. A *primitive recursive relation* is a primitive recursive $R : \mathbb{N}^k \to \mathbb{N}$ valued in $\{0, 1\}$. We also say that a relation $R : \mathbb{N}^k \to \mathsf{Prop}$ *has primitive recursive structure* if it has a characteristic function (as in Section 3.4) χ_R with $\mathsf{PrimRec}(\chi_R)$.

The following constructions are routine.

Theorem 7.3. *The following all have primitive recursive structure.*

(i) the standard ordering $\leq$ on $\mathbb{N}$.

(ii) equality on $\mathbb{N}$.

(iii) the addition function $- + - : \mathbb{N}^2 \to \mathbb{N}$;

(iv) the multiplication function $- \cdot - : \mathbb{N}^2 \to \mathbb{N}$;

(v) the predecessor function with definition

$$\begin{aligned} \mathsf{pred}(0) &= 0, \\ \mathsf{pred}(n+1) &= n; \end{aligned}$$

(vi) truncated subtraction

$$\begin{aligned} n - 0 &= n, \\ n - (k+1) &= \mathsf{pred}(n - k); \end{aligned}$$

(vii) bounded sums: for fixed (primitive recursive) $f : \mathbb{N}^{n+1} \to \mathbb{N}$, and $x : \mathbb{N}^n$,

$$\begin{aligned} \sum_{y<0} f(x, y) &= 0, \\ \sum_{y<(k+1)} f(x, y) &= f(x, k) + \sum_{y<k} f(x, y); \end{aligned}$$

(viii) bounded products: for fixed (primitive recursive) $f : \mathbb{N}^{n+1} \to \mathbb{N}$, and $x : \mathbb{N}^n$,

$$\prod_{y<0} f(x,y) = 1,$$
$$\prod_{y<(k+1)} f(x,y) = f(x,k) \cdot \prod_{y<k} f(x,y);$$

(ix) $\mathsf{sn} : \mathbb{N} \to \mathbb{N}$,

$$\mathsf{sn}(0) = 0,$$
$$\mathsf{sn}(n+1) = 1;$$

(x) $\overline{\mathsf{sn}} : \mathbb{N} \to \mathbb{N}$,

$$\overline{\mathsf{sn}}(0) = 1,$$
$$\overline{\mathsf{sn}}(n+1) = 0;$$

alternatively, $\overline{\mathsf{sn}}(n) = 1 - \mathsf{sn}(n)$*;*

(xi) the factorial function

$$n! = \prod_{k<n} (k+1).$$

Using the above, we can lift primitive recursive structure over logical operations as follows: let P and Q have primitive recursive structure, then we have characteristic functions with primitive recursive structure

$$\begin{aligned}
(p \wedge q)(n) &\stackrel{\text{def}}{=} p(n) \cdot q(n), \\
(p \vee q)(n) &\stackrel{\text{def}}{=} \mathsf{sn}(p(n) + q(n)), \\
(p \to q)(n) &\stackrel{\text{def}}{=} p(n) \leq q(n), \\
(\neg p)(n) &\stackrel{\text{def}}{=} \overline{\mathsf{sn}}(p(n)), \\
\exists(k < n).p(k) &\stackrel{\text{def}}{=} \mathsf{sn}(\sum_{k<n} p(k)), \\
\forall(k < n).p(k) &\stackrel{\text{def}}{=} \prod_{k<n} p(k),
\end{aligned}$$

for (respectively) $P \wedge Q$, $P \vee Q$, $P \to Q$, $\neg P$, $\exists(k \leq n), P(k)$ and $\forall(k \leq n), P(k)$. Then we may show a relation has primitive recursive structure by defining it in terms of known primitive recursive relations. Likewise, we can build primitive recursive structure piecewise

Lemma 7.4. *Let $f, g : \mathbb{N}^k \to \mathbb{N}$ have primitive recursive structure and let $Q : \mathbb{N}^k \to \mathsf{Prop}$ have primitive recursive structure. Then the function $\mathbb{N}^k \to \mathbb{N}$ defined by case analysis as*

$$x \mapsto \begin{cases} f(x) & \textit{if } Q(x); \\ g(x) & \textit{otherwise} \end{cases}$$

has primitive recursive structure

Proof. First define $h : \mathbb{N}^{k+1} \to \mathbb{N}$ by $h(0, x) = g(x)$ and $h(1, x) = f(x)$. That is,

$$h = \mathsf{rec}_{g,f'}$$

where $f'(n, k, x) = f(x)$. Then h has primitive recursive structure. Now define

$$r(x) = h \circ \langle Q(x), \mathsf{pr}_1(x), \ldots, \mathsf{pr}_{n-1}(x) \rangle,$$

which has primitive recursive structure. Now let $x : \mathbb{N}^n$. If $Q(x)$, then $r(x) = h(1, x) = f(x)$ and if $\neg Q(x)$, then $r(x) = h(0, x) = g(x)$. Hence, $r(x)$ is the function defined by case analysis above. $\square$

The last piece required before continuing the standard presentation of the primitive recursive functions is bounded minimization. Let $P : \mathbb{N} \to \mathsf{Prop}$ be a primitive recursive predicate, fix an $n : \mathbb{N}$ and consider the type $\mathsf{min}_{k<n}(P)$

$$\Sigma(k : \mathbb{N}), \big((k < n \times P(k)) + (k = n)\big) \times \Pi(j < k), \neg P(j)),$$

and the map $\mathsf{min}_{k<n}(P) \to \mathbb{N}$ given by the first projection. That is, $\mu_{k<n}P(k)$ is the least value of $k \leq n$ such that $P(k)$ holds, if such exists, and n otherwise.

Lemma 7.5. *For any predicate $P : \mathbb{N} \to \mathsf{Prop}$ with primitive recursive structure and any $n : \mathbb{N}$, the type $\mathsf{min}_{k<n}(P)$ is contractible. Hence, the partial element $(\mathsf{min}_{k<n} P(k), \mathsf{pr}_0)$ is defined.*

Proof. First, show that $\mathsf{min}_{k<n} P(k)$ is inhabited; i.e., that we have $k : \mathbb{N}$ satisfying the predicate

$$Q(k) \overset{\mathsf{def}}{=} \big((k < n \times P(k)) + (k = n)\big) \times \Pi(j < k), \neg P(j)).$$

Since P has primitive recursive structure, it is decidable. Hence, we can do a bounded search to determine

$$(\Sigma(k < n), P(k)) + \Pi(k < n), \neg P(k).$$

In the first case, take the minimum such k, which must exist. In the second, n must satisfy Q.

To show that $\mathsf{min}_{k<n} P(k)$ is a proposition, take any k, j satisfying the predicate Q we must have $k = j$, since otherwise we have $k < j$ (or conversely), and then we must have both $P(k)$ and $\neg P(k)$, a contradiction, and equality on $\mathbb{N}$ is decidable. □

Then we may define for any predicate P with primitive recursive structure

$$\mu_{k<-}(P) : \mathbb{N} \to \mathbb{N}$$

$$\mu_{k<n}(P) \overset{\mathsf{def}}{=} \mathsf{value}(P)(\star)$$

I.e., $\mu_{k<n}(P)$ is the first projection applied to the center of contraction of $\mathsf{min}_{k<n}(P)$.

Theorem 7.6. *If P has primitive recursive structure, then the function $\mu_{k<-}(P)$ has primitive recursive structure.*

Proof. Note that the predicate Q used in the definition of $\mathsf{min}_{k<n} P(k)$ is itself primitive recursive, being built up of bounded quantifiers and basic logical operations. So let q be a primitive recursive term computing Q. Now consider the function $f : \mathbb{N}^2 \to \mathbb{N}$ defined by primitive recursion with

$$f(0,n) = \begin{cases} 0 & \text{if } Q(0), \\ \\ n & \neg Q(0); \end{cases}$$

$$f(k+1,n) = \begin{cases} \mathsf{min}(k+1, f(k,n)) & \text{if } Q(k+1); \\ \\ \mathsf{min}(n, f(k,n)) & \text{if } \neg Q(k+1). \end{cases}$$

Note that f is defined by primitive recursion over the base function $g : \mathbb{N} \to \mathbb{N}$ which takes value 0 if $Q(0)$ and is the identity otherwise, and recursive step given by

$$h(k,m,n) = \begin{cases} \mathsf{min}(k+1, m) & \text{if } Q(k+1); \\ \\ \mathsf{min}(n, m) & \text{if } \neg Q(k+1). \end{cases}$$

So then we see that h has primitive recursive structure as a case analysis, and f has primitive recursive structure as h and g do. Now, consider the value of $f(n-1,n)$: If $n = 0$ or $n = 1$, this is 0. Otherwise, we know that either there is a least $k < n$ with $Q(k)$ or else there is no $k < n$ with $Q(k)$. If there is no such k, then $f(k,n) = n$ for each $k < n$, and so $f(n-1,n) = n$. If there is such a k, then we must have that $f(k,n) = k$ since for $j < k$, we must have $f(j,n) = n$. Then we know that

$$\mathop{\mathsf{min}}_{k<n} P(k) = f(n-1,n),$$

and the function $n \mapsto f(n-1,n)$ is defined by composition from functions with primitive recursive structure. $\square$

We can continue to mimic the classical development of primitive recursion; from here, on there will be no surprises. However, we stop to mention encodings of sequences, since we will use them when working with general computability: we will deal with higher arities via a primitive recursive encoding. Specifically, for each n, we we fix a bijective pairing function $\langle - \rangle : \mathbb{N}^n \to \mathbb{N}$ which is primitive recursive and such that the projection functions $\mathsf{pr}_i : \mathbb{N} \to \mathbb{N}$ are primitive recursive. Defining such functions, and indeed even a primitive recursive function $\langle - \rangle : \sum(n : \mathbb{N}), \mathbb{N}^n \to \mathbb{N}$ is a standard exercise—for $\langle - \rangle : \sum(n : \mathbb{N}), \mathbb{N}^n \to \mathbb{N}$, we can use Gödel's β. Then we define an n-ary function to be computable if its composition with $\langle - \rangle^{-1}$ is computable. Note that by precomposition with $\langle - \rangle$, any function can be viewed as an n-ary function for any n.

In fact, if both $\langle - \rangle : \mathbb{N}^k \to \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$ have primitive recursive structure, then the composite function $f \circ \langle - \rangle : \mathbb{N}^k \to \mathbb{N}$ does as well. A direct converse doesn't type check, but we can do the following: For any $k : \mathbb{N}$ fix a pairing function $\langle - \rangle : \mathbb{N}^k \to \mathbb{N}$ and projection functions $\mathsf{pr}_i : \mathbb{N} \to \mathbb{N}$ all of which have primitive recursive structure and such that

$$\langle \mathsf{pr}_0(n), \ldots, \mathsf{pr}_{k-1}(n) \rangle = n$$

and

$$\mathsf{pr}_i(\langle x_0, \ldots, x_{k-1} \rangle) = x_i.$$

For $f : \mathbb{N}^k \to \mathbb{N}$ with primitive recursive structure define $f' : \mathbb{N} \to \mathbb{N}$ by

$$f' = f \circ \langle \mathsf{pr}_0, \ldots, \mathsf{pr}_{k-1} \rangle$$

so that

$$f'(n) = f(\mathsf{pr}_0\langle n \rangle, \ldots, \mathsf{pr}_{k-1}\langle n \rangle) = f(n).$$

In short, we can encode all primitive recursive functions of k variables as primitive recursive functions of 1 variable. We will tacitly switch between a function of a single variable and a function of k variables when it suits us.

Similarly, we may encode the type $\mathbb{N}+\mathbb{N}$ via functions $\mathsf{inl}, \mathsf{inr} : \mathbb{N} \to \mathbb{N}$, given by, for example, $\mathsf{inl}(n) = 2n$ and $\mathsf{inr}(n) = 2n + 1$. Now given two functions (with p.r. structure) $f, g : \mathbb{N} \to \mathbb{N}$ we can *extend* them along $\mathbb{N} + \mathbb{N}$ to a function $f + g : \mathbb{N} \to \mathbb{N}$ with primitive recursive structure defined by

$$(f+g)(n) = \begin{cases} f(k) & \text{if } n = 2k; \\ g(k) & \text{if } n = 2k+1. \end{cases}$$

This gives us a primitive recursive encoding of $\mathbb{N} + \mathbb{N}$, which we will use to define a notion of computation.

7.2 Recursive machines

We abstract away the details of initializing a Turing machine, and the transition function. The central ideas that captures how a Turing machine computes are

- a Turing machine can be initialized by a simple process;
- checking whether a Turing computation has completed is simple;
- the transition function updating a Turing machine is a simple process.

We capture these properties in the following definition.

Definition 7.7. A *recursive machine*, is a pair $(i, s) : \mathsf{PR}_1 \times \mathsf{PR}_1$. The function i is called the *initialization function*, and we treat it as a function $i : \mathbb{N} \to \mathbb{N}$, while s is called the *transition function*, and we treat it as a function $s : \mathbb{N} \to \mathbb{N} + \mathbb{N}$.

We use RM for the type of recursive machines.

We can evaluate recursive machines via a function

$$\mathsf{eval} : \mathsf{RM} \to (\mathbb{N} \to \mathcal{L}\,\mathbb{N})$$

as follows: Given $(i, s) : \mathsf{RM}$, let $s' : \mathbb{N} + \mathbb{N} \to \mathbb{N} + \mathbb{N}$ be the function

$$s'(\mathsf{inl}\,x) = s(x),$$
$$s'(\mathsf{inr}\,y) = \mathsf{inr}\,y.$$

As we will use this function s', we will abuse notation and refer to s'^k as s^k.

Now define for each $k : \mathbb{N}$ the function $\mathsf{run}_k : \mathsf{RM} \to \mathbb{N} \to (\mathbb{N} + \mathbb{N})$ by

$$\mathsf{run}_k((i, s), x) \overset{\mathsf{def}}{=} s^k(\mathsf{inl}(i(x)))$$

And so we have for fixed $m = (i, s)$,

$$R_m(x, y) \overset{\mathsf{def}}{=} \exists(k : \mathbb{N}), \mathsf{run}_k(m, x) = \mathsf{inr}\, y.$$

Then $\mathsf{eval}(m)$ is the partial function with R_m as its graph. Note that R_m can be defined without truncation using the results of Section 3.4.

The partial function $\mathsf{eval}(m)$ is the function *computed by* m. We will sometimes abuse notation and write $m(x)$ instead of $\mathsf{eval}(m, x)$. For a partial function $f : \mathbb{N} \to \mathcal{L}\,\mathbb{N}$, let $\mathsf{CompStruct}(f)$ be the type of recursive machines computing f:

$$\mathsf{CompStruct}(f) \overset{\mathsf{def}}{=} \Sigma(m : \mathsf{RM}), f = \mathsf{eval}(m).$$

Say that m is a *computability structure* for f or that f has *recursive structure*, when m computes f, i.e., when $(m, -) : \mathsf{CompStruct}(f)$. Note that we have

$$\mathsf{RM} \simeq \Sigma(f : \mathbb{N} \to \mathcal{L}(\mathbb{N})), \mathsf{CompStruct}(f).$$

If $f : \mathbb{N} \to \mathbb{N}$ is an ordinary function, then f has recursive structure when $\eta \circ f : \mathbb{N} \to \mathcal{L}\,\mathbb{N}$ does.

Functions with recursive structure are closed under composition. Indeed, given machines m and n, we can define a composite machine $m; n$ as follows: First define $i' : \mathbb{N} \to \mathbb{N} + \mathbb{N}$ by

$$i'(k) = \mathsf{inl}(i_m k);$$

and a function $s' : \mathbb{N} + \mathbb{N} \to ((\mathbb{N} + \mathbb{N}) + \mathbb{N})$ by

$$
\begin{aligned}
s'(\mathsf{inl}\, k) &= \mathsf{inl}(\mathsf{inl}\, y) && \text{if } s_m(k) = \mathsf{inl}\, y; \\
s'(\mathsf{inl}\, k) &= \mathsf{inl}(\mathsf{inr}(i_n y)) && \text{if } s_m(k) = \mathsf{inr}\, y; \\
s'(\mathsf{inr}\, k) &= \mathsf{inl}(\mathsf{inr}\, y) && \text{if } s_n(k) = \mathsf{inl}\, y; \\
s'(\mathsf{inr}\, k) &= \mathsf{inr}\, y && \text{if } s_n(k) = \mathsf{inr}\, y;
\end{aligned}
$$

Then the three components in the codomain $(\mathbb{N} + \mathbb{N}) + \mathbb{N}$ correspond (from left to right) to "we are still computing m"; "we have computed m and are now computing n"; and "we have finished computing the composite". By post composing with a primitive recursive bijection $c : \mathbb{N} + \mathbb{N} \to \mathbb{N}$, we get $(c \circ i', c \circ s')$ as a recursive machine.

Lemma 7.8. *If m and n compute f and g respectively, then $m; n$ computes $g \mathbin{\square} f$.*

Proof. We show that the relation $R_m; R_n$ used in the definition of eval is the same as the relation $R_{m;n}$.

Suppose that $(R_m; R_n)(x, z)$. That is,

$$\exists(y : \mathbb{N}), \exists(k : \mathbb{N}), (s_m^k(i_m x) = \mathsf{inr}(y)) \times (\exists(j : \mathbb{N}), s_n^k(i_n y) = z).$$

We could give a non-truncated equivalent type, but this is not necessary here: by currying and properties of truncation, to get $R_{m;n}(x, z)$ it is enough to show

$$\Pi(y : \mathbb{N}), \big(\Sigma(k : \mathbb{N}), s_m^k(x) = \mathsf{inr}(y)\big) \to \big(\Sigma(j : \mathbb{N}), s_n^j(y) = z\big) \to R_{m;n}(x, z).$$

The definition of $m; n$ was chosen specifically to satisfy this.

Conversely, if $R_{m;n}(x, z)$, then take k least such that $s'(\mathsf{inl}(i_m x)) = \mathsf{inr}\, y$ for some y, so that $s_m^k(i_m x) = \mathsf{inr}\, y'$ with $i_n(y') = y$. Then we must have $s_n^j(y) = z$ for some j, since $R_{m;n}(x, z)$. $\square$

For convenience, we will often describe recursive machines in an informal way, making reference to “configurations”, “continuing”, and “halting”. As an example, we again describe the machine for $m; n$:

To initialize $m; n$, initialize m. For the transition function, the input is in one of two states:

1. the input represents a partially computed output of m;
2. the input represents a partially computed output of n.

In each case, do the following:

1. if one step of m completes the computation of $m(x)$ with value y, then output the initialization of n at y in state (2). Otherwise, take one more step of m and continue in state (1).
2. if one step of n completes the computation of $n(x)$ with value y, then output y and halt. Otherwise, take one more step of m and continue in state (2).

7.3 Recursive predicates and minimization

Definition 7.9. A *recursive relation* is a total function $R : \mathbb{N}^k \to \mathbb{N}$ valued in $\{0, 1\}$ with a computability structure. We also say that a relation $R : \mathbb{N}^k \to \mathsf{Prop}$ *has recursive structure* if it has a characteristic function χ_R with $\mathsf{CompStruct}(\chi_R)$.

Definition 7.10. We have a *minimization operator* for partial functions, $\mu : (\mathbb{N} \to \mathcal{L}\,\mathbb{N}) \to \mathcal{L}\,\mathbb{N}$, with

$$\mathsf{defined}(\mu f) \overset{\mathsf{def}}{=} \Sigma(k : \mathbb{N}), f(k) = \eta 0 \times \Pi(j < k), \Sigma(n : \mathbb{N}), f(j) = \eta(n+1),$$

and $\mathsf{value}(\mu f) \overset{\mathsf{def}}{=} \mathsf{pr}_1$. More generally, we may define the *minimization* $\mu y.R(y) : \mathcal{L}\,\mathbb{N}$ *of a predicate* $R : \mathbb{N} \to \mathsf{Prop}$

$$\mathsf{defined}(\mu y.R(y)) \overset{\mathsf{def}}{=} \Sigma(y : \mathbb{N}), R(y) \times \Pi(x < y), \neg R(x).$$

and $\mathsf{value}(\mu y.R(y)) \overset{\mathsf{def}}{=} \mathsf{pr}_1$.

When there is a least such y, this does indeed choose it. Moreover, the extent of $\mu y.R(y)$ is a proposition. That is

Lemma 7.11. *For any predicate $R : \mathbb{N} \to \mathsf{Prop}$, the type $\Sigma(y : \mathbb{N}), R(y) \times \Pi(x < y), \neg R(x)$ is a proposition.*

Proof. For any predicate $R : \mathbb{N} \to \mathsf{Prop}$ and any $y : \mathbb{N}$ we have that $R(y) \times \Pi(x \leq y), \neg R(x)$ is a proposition, as a product of propositions, so we need only check the first component. That is we need to see that $m = n$ whenever we have witnesses $w : R(m) \times \Pi(x < m), \neg R(x)$ and $v : R(n) \times \Pi(x < n), \neg R(x)$. As the ordering on $\mathbb{N}$ is decidable, we have either $m < n$, $m = n$ or $n < m$. If $m < n$, then we have $\neg R(m)$ by v and $\mathrm{R}(m)$ by w, a contradiction. Symmetrically, we cannot have $n < m$, and so $n = m$. $\square$

Note that if there is a unique $y : \mathbb{N}$ satisfying a predicate, then we can find y by minimization. Then expanding the definition of eval and μ, we see that for any recursive machine m, we have

$$\mathsf{eval}(m)(x) = \mu y. \exists (k : \mathbb{N}), \mathsf{run}_k(m, x) = \mathsf{inr}\, y.$$

Using this fact we can prove that the minimization of a relation with primitive recursive structure has recursive structure.

Theorem 7.12. *If $R : \mathbb{N}^2 \to \mathsf{Prop}$ has primitive recursive structure, then $\lambda x.\mu y.R(x, y)$ has recursive structure.*

Proof. Define

$$i(x) \stackrel{\mathsf{def}}{=} \langle x, 0 \rangle;$$

and

$$s(\langle x, n \rangle) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{inr}(n) & \text{if } R(x, n); \\ \mathsf{inl}\langle x, n+1 \rangle & \text{otherwise.} \end{cases}$$

Pairing has primitive recursive structure, and case analysis on a primitive recursive predicate has a primitive recursive structure, so we know i and s have primitive recursive structure. Hence $m = (i, s)$ is a recursive machine. We need to see that $\mathsf{eval}(m)(x) = \mu y.R(x, y)$. We have that $\mathsf{run}_k(m, x) = \mathsf{inl}\langle x, k+1 \rangle$ until $R(x, k)$. Then $\mathsf{run}_k(m, x) = \mathsf{inr}\, k$ precisely when k is least such that $R(x, k)$. $\square$

It is tempting to use the same argument to show that this works for any *recursive* predicate, but we need to be more careful: the function s defined in the proof is only primitive recursive because R is. For a general recursive predicate, more work needs to be done. The tools required are discussed in the next section.

7.4 The Normal Form Theorem

In this section we prove a version of the normal form theorem for recursive machines. That is, we define encodings of recursive machines, such that we may define Kleene's predicate $\mathsf{T} : \mathbb{N}^3 \to \mathsf{Prop}$ and function $\mathsf{U} : \mathbb{N} \to \mathbb{N}$. The meaning of $\mathsf{T}(m, x, y)$ is that y represents a full computation trace of the application of the function encoded by m to x—such a y exists only when $m(x)$ produces a value. The function U then extracts the value produced by $m(x)$. We will then prove that the minimization of an arbitrary predicate with computability structure defines a function with computability structure.

In order to prove this, we first prove a version for primitive recursive combinators.

Theorem 7.13 (Kleene's normal form theorem for Primitive recursive functions)**.** *There is an injection* $\mathsf{encode} : \mathsf{PR} \to \mathbb{N}$, *a predicate* $\mathsf{T}' : \mathbb{N}^3 \to \mathsf{Prop}$ *with primitive recursive structure, and a function* $\mathsf{U}' : \mathbb{N} \to \mathbb{N}$ *with primitive recursive structure such that for any primitive recursive combinator* t *and any* $x : \mathbb{N}$ *we have that* $\Sigma(y : \mathbb{N}), \mathsf{T}'(\mathsf{encode}(t), x, y)$ *is contractible, and moreover*

$$\mathsf{app}(t, x) = \mathsf{U}'(y),$$

whenever y *is such that* $\mathsf{T}'(\mathsf{encode}(t), x, y)$.

Theorem 7.14 (Kleene's normal form theorem for recursive machines)**.** *There is an injection* $\mathsf{encode} : \mathsf{RM} \to \mathbb{N}$, *a predicate* $\mathsf{T} : \mathbb{N}^3 \to \mathcal{U}$ *with primitive recursive structure, and a function* $\mathsf{U} : \mathbb{N} \to \mathbb{N}$ *with primitive recursive structure such that for any recursive machine* m *and any* $x : \mathbb{N}$ *we have that* $\Sigma(y : \mathbb{N}), \mathsf{T}(\mathsf{encode}(m), x, y)$ *is a proposition, and moreover*

$$\mathsf{eval}(m, x) = ((\Sigma(y : \mathbb{N}), \mathsf{T}(\mathsf{encode}(m), x, y)), \mathsf{U} \circ \mathsf{pr}_1).$$

Recall that we have a primitive recursive onto coding of finite sequences, along with projection and length functions. We use $\langle a_0, \ldots, a_n \rangle$ for the encoding of the sequence $(a_0, \ldots, a_n)$, and $+\!\!+$ for concatenation. If x is a number representing a sequence, then $(x)_i$ is the i-th coordinate of x.

Now, we define an interpretation

$$\begin{aligned}
\mathsf{encode} &: \mathsf{PR} \to \mathbb{N} \\
\mathsf{encode}(\mathsf{s}) &= \langle 0 \rangle, \\
\mathsf{encode}(\mathsf{p}^n_k) &= \langle 1, n, k \rangle, \\
\mathsf{encode}(\mathsf{c}^n_k) &= \langle 2, n, k \rangle, \\
\mathsf{encode}(f\langle g_1, \dots, g_n \rangle) &= \langle 3 \rangle \mathbin{+\!\!+} \mathsf{encode}(f) \mathbin{+\!\!+} \mathsf{encode}(g_1) \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathsf{encode}(g_n), \\
\mathsf{encode}(\mathsf{r}_{f,g}) &= \langle 4 \rangle \mathbin{+\!\!+} \mathsf{encode}(f) \mathbin{+\!\!+} \mathsf{encode}(g).
\end{aligned}$$

To arrive at T' and U', we first define a type of computation trees. Each node of the tree will have 3 labels:

- (The code of) a primitive recursive combinator,
- an integer x, representing a sequence of inputs,
- an integer z representing an output.

We define

1. $[\mathsf{s}, x, y]$ is a leaf computation tree;
2. $[\mathsf{p}^n_k, x, y]$ is a leaf computation tree;
3. $[\mathsf{c}^n_k, x, y]$ is a leaf computation tree;
4. $[f\langle g_1, \dots, g_n \rangle, x, y]$ is the root of a computation tree with $n + 1$ branches;
5. $[\mathsf{r}_{f,g}, \langle 0, x \rangle, y]$ is the root of a computation tree with 1 branch;
6. $[\mathsf{r}_{f,g}, \langle n + 1, x \rangle, y]$ is the root of a computation tree with 2 branches;

A computation tree t is *correct* when

1. the root of t is $[\mathsf{s}, x, x + 1]$ for some $x : \mathbb{N}$;
2. the root of t is $[\mathsf{p}^n_k, x, k]$ for some $x : \mathbb{N}$;
3. the root of t is $[\mathsf{c}^n_k, x, (x)_0]$, for some $x : \mathbb{N}$;
4. the root of t is $[f\langle g_1, \dots, g_n \rangle, x, y]$, the branches of t are all correct, and
   - the 0-th branch has root $[f, \langle z_1, \dots, z_n \rangle, y]$, and
   - for $0 < k \leq n$, the k-th branch has root $[g_k, x, z_k]$;

5. the root of t is $[\mathsf{r}_{f,g}, \langle 0 \rangle +\!\!+\, x, y]$, the branch is correct and has label $[g, x, y]$.
6. the root of t is $[\mathsf{r}_{f,g}, \langle n+1 \rangle +\!\!+\, x, y]$, both branches are correct, and
    - the 0-th branch has root $[\mathsf{r}_{f,g}\langle n \rangle +\!\!+\, x, z]$, and
    - the first branch has root $[f, \langle n \rangle +\!\!+\, x +\!\!+\, \langle z \rangle, y]$;

In short, a computation tree with root $[t, x, y]$ is correct precisely when $\mathsf{app}(t, x) = y$.

Now we can encode a tree t with root label $[c, x, y]$ and branches $\{t_i\}_{i<n}$ as

$$\hat{t} = \langle \langle \mathsf{encode}(c), x, y \rangle, \hat{t}_0, \ldots, \hat{t}_{n-1} \rangle.$$

By expanding the definition of the coding of sequences, and using the definition of "correct" above, we get a primitive recursive predicate $\mathsf{C}(x)$ expressing "x encodes a correct computation tree." Now we may define

$$\mathsf{T}'(e, x, y) \stackrel{\mathsf{def}}{=} \mathsf{C}(y) \wedge \big(((y)_0)_0 = e\big) \wedge \big(((y)_0)_1 = x\big).$$

and

$$\mathsf{U}'(y) \stackrel{\mathsf{def}}{=} ((y)_0)_2,$$

so that we can define $\{-\}_{\mathsf{PR}}$ to be

$$\{n\}_{\mathsf{PR}}(x) = \mathsf{U}'(\mu_y.\, \mathsf{T}'(n, x, y)).$$

We need to see that for any $t : \mathsf{PR}$ and $x : \mathbb{N}$

$$\mathsf{app}(t, x) = \{\mathsf{encode}(t)\}_{\mathsf{PR}}(x),$$

i.e., that

$$\mathsf{app}(t, x) = \mathsf{U}'(\mu_y.\, \mathsf{T}'(n, x, y)).$$

But we've constructed T' such that

$$\mathsf{T}'(n, x, y) \Leftrightarrow \mathsf{eval}(t, x) = y.$$

Note that for a fixed k and recursive machine m, $\mathsf{run}_k(m) : \mathbb{N} \to \mathbb{N}$ has primitive recursive structure. In fact, we can do better.

Lemma 7.15. *The function*

$$\mathsf{source} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

taking a number k and $\langle \mathsf{encode}(i), \mathsf{encode}(s)\rangle$ to $\mathsf{encode}(\mathsf{run}_k(i, s))$ has primitive recursive structure.

Proof. Straightforward induction on k and the combinators for i and s. □

Then we may use T' and U' to define T and U for recursive machines:

$$\mathsf{T}(e, x, \langle k, y\rangle) = \mathsf{T}'(\mathsf{source}(k, e), x, y).$$

and

$$\mathsf{U}(\langle k, y\rangle) = \mathsf{U}'(y).$$

This result gives us the Kleene-bracket function $\{-\} : \mathbb{N} \to (\mathbb{N} \to \mathcal{L}\,\mathbb{N})$, defined by

$$\{e\}(x) \stackrel{\mathsf{def}}{=} \mathcal{L}\,\mathsf{U}(\mu y.\, \mathsf{T}(e, x, y)).$$

In order to be able to apply $\{-\}$ to a partial natural number, we may Kleisli extend $\{-\}$ in the first component. We will abuse notation and write also $\{-\} : \mathcal{L}\,\mathbb{N} \to (\mathbb{N} \to \mathcal{L}\,\mathbb{N})$ for this map.

Theorem 7.16. *If $R : \mathbb{N} \to \mathsf{Prop}$ has recursive structure, then case analysis on R has recursive structure. Explicitly, if R is a recursive predicate and $e, e' : \mathsf{RM}$ then the function $f : \mathbb{N} \to \mathcal{L}\,\mathbb{N}$ defined* *by*

$$f(x) = \begin{cases} e(x) & \text{if } R(x) \\ \\ e'(x) & \text{if } \neg R(x) \end{cases}$$

has recursive structure.

Proof. The initialization function initializes R for the transition function with 3 states:

1. the input represents a pair (x, y), where x is the input to f, and y is a partially computed value of $R(x)$;
2. the input represents a partially computed value of $e(x)$;
3. the input represents a partially computed value of $e'(x)$.

The transition function does the following on y in each case

1. if one step of $s_R(\mathsf{pr}_1\, y)$ completes the computation of $R(x)$, then check if $R(x)$ returns 1 or 0, and return the initialization of e or e' at x accordingly, and continue in the corresponding sates. If not, then return $(x, s_R(\mathsf{pr}_1\, y))$, continuing in state (1);
2. if one step of $s_e(y)$ completes the computation of $e(x)$, then return this and halt, otherwise take one step and continue in state (2);
3. if one step of $s_{e'}(y)$ completes the computation of $e'(x)$, then return this and halt, otherwise take one step and continue in state (3).

As this returns $e(x)$ when $R(x) = 1$ and $e'(x)$ when $R(x) = 0$, this machine is correct for f. □

Theorem 7.17. *If $R : \mathbb{N}^k \to \mathsf{Prop}$ has recursive structure, then $\lambda x.\mu y.R(x, y)$ has recursive structure.*

Proof. The initialization function initializes x to $\langle x, 0, i_R(x, 0)\rangle$. The transition function does the following on input $\langle x, y, c\rangle$:

If one step of R applied to c finishes the computation of $R(x, y)$ with output 1, then return y and halt. Otherwise, if one step of R applied to c finishes the computation of (x, y) with output 0, then return $\langle x, y + 1, i_R(x, y + 1)\rangle$ and continue. Finally, if neither of these cases hold, then we must continue computing R at c, so output $\langle x, y, s_R(c)\rangle$ and continue. □

Theorem 7.18. *If $f : \mathbb{N}^{n+2} \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$ have recursive structure, then $\mathsf{rec}_{f,g}$ has recursive structure.*

Thanks to Theorem 7.17, the classical proof of elimination of primitive recursion via Gödel's β function also gives this result.

Proof. Define

$$t(x,y) = \mu z.\big(\beta(z,0) = g(x) \wedge (\forall i < y.\beta(z,i+1) = f(x,i,\beta(z,i)))\big).$$

Then we have

$$\mathsf{rec}_{f,g}(x,y) = \beta(t(x,y),y). \qquad \square$$

Using the above, we get the most difficult cases of the classical characterization of Turing computability in terms of μ-recursiveness.

Theorem 7.19. 1. *All constant functions $\mathbb{N}^k \to \mathbb{N}$ have recursive structure;*

2. *the successor function has recursive structure;*
3. *all projections have recursive structure;*
4. *if $f : \mathbb{N}^k \to \mathbb{N}$ has recursive structure and $g_i : \mathbb{N}^n \to \mathbb{N}$ has recursive structure for each $i < k$, then $\lambda x.f(g_0(x),\ldots,g_{k-1}(x))$ has recursive structure*
5. *if $f : \mathbb{N} \rightharpoonup \mathbb{N}$ and $g : \mathbb{N} \rightharpoonup \mathbb{N}$ have recursive structure, then $g \,\square\, f$ has recursive structure;*
6. *If $R : \mathbb{N}^2 \to \mathsf{Prop}$ has recursive structure, then $\lambda x.\mu y.R(x,y)$ has recursive structure.*

Proof. Parts (1), (2) and (3) are immediate; part (5) is Theorem 7.18; part (6) is Theorem 7.17. For (4), we use a similar trick as for part (5): let

$$t(x) = \mu z.\big(\beta(z,0) = g_0(x) \wedge \ldots \wedge \beta(z,k-1) = g_{k-1}(x)\big),$$

and then we have $(t;f)(x) = \lambda x.f(g_0(x),\ldots,g_{k-1}(x)))$. $\square$

There is one last result to prove using the techniques of this section, before we can develop recursion theory without worrying about the details of how to encode computable functions

Theorem 7.20 (S^m_n Theorem)**.** *There is a primitive recursive $\mathsf{s} : \mathbb{N}^2 \to \mathbb{N}$ such that for all $e,x,y : \mathbb{N}$ we have $\{\mathsf{s}(e,x)\}(y) = \{e\}\langle x,y\rangle$.*

Proof. Let e encode a recursive machine computing $f : \mathbb{N} \rightharpoonup \mathbb{N}$, let $x : \mathbb{N}$ and define $f' : \mathbb{N} \rightharpoonup \mathbb{N}$ by $f'(y) = f(\langle x, y\rangle)$. By manipulating the tree defining e, we can get a machine m computing f', which we can encode as e'. This manipulation is primitive recursive in e and x, since it amounts to simple operations on finite sequences. Take $\mathsf{s}(e, x) = e'$. Then

$$\{\mathsf{s}(e, x)\}(y) = \{e'\}(y) = \{e\}\langle x, y\rangle. \qquad \square$$

7.5 Basic Recursion Theory

We are now able to show how to develop basic recursion theory via recursive machines. We roughly follow the presentation in Odifreddi [62]; in fact, the proofs given by Odifreddi for the results stated here translate to our framework with only minor adjustment.

Rogers' Fixed Point Theorem and Kleene's Second Recursion Theorem can be proved in the standard way.

Theorem 7.21 (Rogers' fixed point theorem)**.** *For any total $f : \mathbb{N} \to \mathbb{N}$ with recursive structure, we have $n : \mathbb{N}$ with $\{n\} = \{f(n)\}$.*

Proof. Let $g : \mathbb{N} \to \mathbb{N}$ be defined by

$$g\langle x, y\rangle \overset{\mathsf{def}}{=} \{\{x\}(x)\}(y).$$

Note that we are Kleisli-extending along the first argument of $\{-\}(-)$ to make sense of $\{\{x\}(x)\}(y)$. As this has recursive structure, it has a code $e_g : \mathbb{N}$. Define $h(x) = \mathsf{s}(e_g, x)$, and consider $e = \mathsf{encode}(f \circ h)$ and $n = h(e)$. Then we have

$$\{n\}y = \{h(e)\}y = \{\mathsf{s}(e_g, e)\} = \{e_g\}\langle e, y\rangle = \{\{e\}(e)\}(y) = \{f(h(e))\}(y) = \{f(n)\}(y). \qquad \square$$

Theorem 7.22 (Kleene's Second Recursion Theorem)**.** *For any function f with recursive structure we have $p : \mathbb{N}$ such that*

$$\Pi(y : \mathbb{N}), (\{p\}(y) = f(\langle p, y\rangle)).$$

Proof. Let $g : \mathbb{N} \to \mathbb{N}$ be defined by

$$g(x) \stackrel{\mathsf{def}}{=} \mathsf{s}\langle e_f, x\rangle,$$

where $e_f : \mathbb{N}$ is a code of the recursive structure for f. This function g has primitive recursive structure, so g is a total function with recursive structure. By Rogers' fixed point theorem, there is then a p such that $\{p\} = \{f(p)\}$. Then,

$$\{p\}(y) = \{f(p)\}(y) = \{g(p)\}(y) = \{\mathsf{s}(e_f, p)\}(y) = f(\langle p, y\rangle).$$ □

Definition 7.23. A *recursive enumeration* of a subset $A : \mathbb{N} \to \mathsf{Prop}$ is a recursive machine m such that $\mathsf{range}(\mathsf{eval}(m)) = A$.

Theorem 7.24. *A subset A of $\mathbb{N}$ has recursive structure if and only if it is complemented and both A and its complement have recursive enumerations.*

Proof. It is immediate that recursive sets are complemented, r.e. and co-r.e. so consider A complemented with f witnessing that A is r.e. and g witnessing that A is co-r.e. according to the characterization in Theorem 7.25. To see that A is recursive, consider the ATM with initialization function $i(n) = \langle i_f(n), i_g(n)\rangle$ and step function

$$s(\langle x, y\rangle) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{inl}\langle n, m\rangle & \text{if } s_f(x) = \mathsf{inl}\, n, s_g(x) = \mathsf{inl}\, m, \\ \mathsf{inr}\, 1 & \text{if } s_f(x) = \mathsf{inr}\, n, \\ \mathsf{inr}\, 0 & \text{if } s_g(x) = \mathsf{inr}\, m. \end{cases}$$

It is clear that this computes the characteristic function of A. We require A to be complemented for this case analysis to be exhaustive. □

Equivalence at the type level, rather than logical equivalence, seems impossible, as we would need to ensure that we respect changes in recursive structure.

Likewise we have the classical characterization of recursively enumerable sets.

Theorem 7.25. *A subset A of $\mathbb{N}$ has a recursive enumeration iff it is the domain of a function with recursive structure.*

Proof. Let f have recursive structure such that $\mathsf{defined}\, f(x) \Leftrightarrow A(x)$. Define $f' : \mathbb{N} \rightharpoonup \mathbb{N}$ by

$$f'(x) \stackrel{\mathsf{def}}{=} (\eta \circ c_x)^\sharp(fx).$$

f' has recursive structure as the composition of functions with computable structure. Moreover we have

$$f'(x) = (\mathsf{defined}(f(x)), \lambda y.x),$$

so that $\mathsf{defined}\, f'(x) = \mathsf{defined}\, f(x)$. Then we have that $A(x) \Leftrightarrow \mathsf{defined}\, f'(x)$.

For the other direction, let g have recursive structure m and $\big(\exists(x : \mathbb{N}), g(x) = \eta n\big) = A(n)$ for all n. Define $h : \mathbb{N} \to \mathbb{N} + \mathbb{N}$ by $h(\langle k, y\rangle) = \mathsf{run}_k(m, y)$. We know that this has primitive recursive structure, so define $j : \mathbb{N} \to \mathbb{N}$ to be

$$j(x) \stackrel{\mathsf{def}}{=} \mu z.h(z) = \mathsf{inr}\, x.$$

It is then easy to check that j is defined exactly on the image of g. Let $\mathsf{defined}(j(x))$. Then there are (k, y) such that $\mathsf{run}_k(m, y) = \mathsf{inr}\, j(x)$. So we have $m(y) = j(x)$. Conversely, let x be in the image of g. That is,

$$\exists(y : \mathbb{N}), \exists(k : \mathbb{N}), \mathsf{run}_k(m, y) = x.$$

But then $\exists(z : \mathbb{N}), g(z) = x$, and from this we see $j(x)$ is defined. $\square$

A slightly better classical characterization requires some modification to be true constructively.

Theorem 7.26. *If A is a subset of $\mathbb{N}$ with $a : A$, then the following are logically equivalent.*

1. *There is a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ with recursive structure whose domain is A.*
2. *There is a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ with recursive structure whose range is A.*
3. *There is a total function $f : \mathbb{N} \to \mathbb{N}$ with recursive structure whose range is A.*
4. *There is a total function $f : \mathbb{N} \to \mathbb{N}$ with primitive recursive structure whose range is A.*

Proof. $(1 \Leftrightarrow 2)$ is Theorem 7.25.

$(2 \Rightarrow 3)$ Let $f : \mathbb{N} \rightharpoonup \mathbb{N}$ be computable with image A, and let $a : A$. Define

$$g(\langle y, k\rangle) = \begin{cases} x & \text{if } \mathsf{run}_k(f, y) = \mathsf{inr}\, x, \\ a & \text{otherwise.} \end{cases}$$

$(2 \Rightarrow 4)$ In fact, the function g above is primitive recursive, as a case analysis over a primitive recursive predicate.

$(4 \Rightarrow 3)$ and $(3 \Rightarrow 2)$ are obvious. $\square$

7.6 Discussion

Except for nuances in the way the results are stated, the results here are the classical ones, and better discussion of their history and importance can be found in more complete texts on computability theory. Instead, let's focus on the goals of this chapter. As stated in the introduction, the goal was not to give a complete constructive account of computability theory, but to examine the notion of partiality in a univalent setting, and to compare computability as structure with computability as property in a setting where we can formally distinguish them. On the first point, we hope the above gives a satisfying enough account to believe that our notion of partiality is robust enough to handle computability theory. It is worth remarking on the fact that when taking partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$ to be functions $\mathbb{N} \to \mathcal{L}(\mathbb{N})$, Kleene equality between partial functions is simply equality.

The distinction between computability as structure and computability as property justifies the nonstandard and sometimes awkward statement of results: we need to keep computable functions separate from functions equipped with computability structure.

The only remaining comment is on the use of recursive machines. The aim of introducing recursive machines was to minimize the number of "moving parts", while capturing the intuitive clarity of Turing machines. Moreover, since minimization is not a total operation, and the primitive recursive combinators all give total functions, it felt natural to separate minimization. Once this separate is made, there is no reason besides history to insist on a minimization operator as the way to introduce fixed-point recursion.

CHAPTER 8

Computability as property

In Chapter 7, we developed a notion of functions *with* computable structure, as those functions which are computed by given recursive machines. Here we turn to computability as property. Since most of the results of this section follow by functoriality of truncation from corresponding results about computability as property, this chapter is quite short. Sections 8.1, 8.2, and 8.3 summarize these results. Section 8.4 contains the proof of the undecidability of the halting problem, and leverages this to define a non-computable partial function, Turing's diagonal function. As all total functions which we can define in the empty context (i.e., without additional assumptions) are computable and constructive intuition suggests that our constructively definable objects ought to be computable; it is not obvious that non-computable partial functions ought to be definable. Their existence means that we cannot simply extend Church's thesis to partial functions in a consistent way. Instead we must weaken the statement that all partial functions are computable. We will chose to do so by restricting the class of computable functions using machinery from Chapter 5, but we will put that off until Chapter 9. In the meantime we give restriction of the partial functions, which is explicitly tied to computation (Section 8.5). This restriction is more concrete than the Rosolini partial functions, since it is tied explicitly to the definition of computability, but this explicit connection to computability makes it less suited for our purposes, since one of our goals is an abstract notion of partial function which could be a surrogate for the computable partial functions.

8.1 Primitive recursive functions

Definition 8.1. The predicate $\mathsf{isPR}_n : (\mathbb{N}^n \to \mathbb{N}) \to \mathcal{U}$ is

$$\mathsf{isPR}_n(f) \stackrel{\mathsf{def}}{=} \|\mathsf{PrimRec}_n(f)\| \,.$$

We say that f *is primitive recursive* when $\mathsf{isPR}(f)$.

Then the primitive recursive functions are the image of the primitive recursive combinators under app.

Definition 8.2. A predicate $R : \mathbb{N}^n \to \mathsf{Prop}$ is *primitive recursive* if it has a primitive recursive characteristic function, and write $\mathsf{isPR}(R)$ for the type of primitive recursive characteristic functions of R.

$$\mathsf{isPR}(R) \stackrel{\mathsf{def}}{=} \Sigma(f : \mathbb{N}^n \to \mathbb{N}), \mathsf{isPR}(f) \times (f(n) = 0 \Leftrightarrow \neg R(n)) \times (f(n) = 1 \Leftrightarrow R(n))$$

As characteristic functions for predicates are unique when they exist, $\mathsf{isPR}(R)$ is a proposition.

As the basic functions of Theorem 7.3 have primitive recursive structure, we have

Theorem 8.3. *The following are all primitive recursive*

(i) *the standard ordering $\leq$ on $\mathbb{N}$,*

(ii) *equality on $\mathbb{N}$,*

(iii) *the addition function $- + - : \mathbb{N}^2 \to \mathbb{N}$,*

(iv) *the multiplication function $- \cdot - : \mathbb{N}^2 \to \mathbb{N}$,*

(v) *the predecessor function,*

(vi) *truncated subtraction,*

(vii) *bounded sums,*

(viii) *bounded products,*

(ix) sn*,*

(x) $\overline{\mathsf{sn}}$*,*

(xi) *the factorial function.*

Similarly, the primitive recursive predicates are closed under the logical operations.

Theorem 8.4. *If P and Q are primitive recursive, then so are $P \wedge Q$, $P \vee Q$, $P \to Q$, $\neg P$, $\exists(k \leq n), P(k)$ and $\forall(k \leq n), P(k)$.*

8.2 Computable functions

Definition 8.5. The predicate $\mathsf{isComputable} : (\mathbb{N} \to \mathcal{L}\,\mathbb{N}) \to \mathcal{U}$ is defined by

$$\mathsf{isComputable}(f) \stackrel{\mathsf{def}}{=} \|\mathsf{CompStruct}(f)\|\,.$$

The partial function f is *computable* when $\mathsf{isComputable}(f)$.

The type Comp is the type of all computable partial functions:

$$\mathsf{Comp} \stackrel{\mathsf{def}}{=} \Sigma(f : \mathbb{N} \to \mathcal{L}\,\mathbb{N}), \mathsf{isComputable}(f)$$

Then Comp is the image of RM under eval.

The distinction between $\mathsf{CompStruct}(f)$ and $\mathsf{isComputable}(f)$ is necessary. By the enumeration theorem for primitive recursive functions, we have a map

$$(\Sigma(f : \mathbb{N} \rightharpoonup \mathbb{N}), \mathsf{CompStruct}(f)) \to \mathbb{N},$$

given by $(i, s) \mapsto \langle e_i, e_s \rangle$, which is easily seen to be an embedding.

On the other hand, we cannot expect there to be an en embedding $F : \mathsf{Comp} \to \mathbb{N}$.

Theorem 8.6. *If there is an embedding $F : \mathsf{Comp} \to \mathbb{N}$, then equality between computable functions is decidable. In particular, an embedding $F : \mathsf{Comp} \to \mathbb{N}$ implies WLPO for computable functions.*

Proof. Let $F : \mathsf{Comp} \to \mathbb{N}$ be an embedding, and let $f, g : \mathsf{Comp}$, so $(f = g) \simeq (F(f) = F(g))$. As $\mathbb{N}$ has decidable equality, $F(f) = F(g)$ is decidable.

Then taking g to be $\lambda x.0$, which we know to be computable, we have that it is decidable whether $f(x) = \lambda x.0$. $\square$

We do not expect to be able to decide whether two computable functions are equal, as then the halting problem would be a decidable predicate. Decidable means complemented rather than recursive here, but we expect any predicate which can be proved to be complemented in our system to be recursive. We will re-examine this situation more closely in Chapter 9.

A consequence of Theorem 8.6 is that we also cannot even expect to have an embedding

$$\mathsf{Comp} \to \Sigma(f : \mathbb{N} \rightharpoonup \mathbb{N}), \mathsf{CompStruct}(f).$$

That is, we have no way of finding a program for a function just by knowing one exists.

However, many basic facts about functions with computable structure also apply to computable functions.

Theorem 8.7. *If f and g are computable, then so is $g \square f$.*

Proof. By functoriality of truncation (Lemma 2.14), we know

$$\big(\mathsf{CompStruct}(f) \to \mathsf{CompStruct}(g) \to \mathsf{CompStruct}(g \square f)\big) \to$$
$$\mathsf{isComputable}(f) \to \mathsf{isComputable}(g) \to \mathsf{isComputable}(g \square f),$$

and the antecedent of this implication is Theorem 7.8. □

Theorem 8.8. *If $R : \mathbb{N}^2 \to \mathsf{Prop}$ is primitive recursive, then*

$$\mathsf{isComputable}(\lambda x.\mu y.R(x, y)).$$

Proof. Functoriality of truncation and Theorem 7.12. □

Definition 8.9. A predicate $R : \mathbb{N}^k \to \mathcal{U}$ is recursive when it has a computable characteristic function.

Theorem 8.10. *If $R : \mathbb{N}^k \to \mathcal{U}$ is recursive, and e, e' are computable, then the function $f : \mathbb{N}^k \to \mathbb{N}$ defined by*

$$f(x) = \begin{cases} e(x) & \text{if } R(x) \\ e'(x) & \text{if } \neg R(x) \end{cases}$$

is also recursive

Proof. Functoriality of truncation from Theorem 7.16. □

Theorem 8.11. *If $R : \mathbb{N}^k \to \mathsf{Prop}$ is recursive, then $\lambda x.\mu y.R(x, y)$ is computable.*

Proof. Functoriality of truncation from Theorem 7.17. □

Theorem 8.12. *If $f : \mathbb{N}^{n+2} \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$ are computable, then $\mathsf{rec}_{f,g}$ is computable.*

Proof. Functoriality of truncation from Theorem 7.18. □

8.3 Recursion theory

Most of the results of Section 7.4 are results that deal explicitly with machines, as such, they do not lift directly to computable functions. However, the theory the results allow can be lifted to computable functions. In particular, we have T, U and $\{-\}$, which we saw have recursive structure, and so these functions are computable. From this, we can prove Roger's fixed point theorem and Kleene's second recursion theorem for computable functions.

Theorem 8.13. *For any total computable function $f : \mathbb{N} \to \mathbb{N}$,*

$$\exists (n : \mathbb{N}), \{n\} = \{f(n)\}.$$

Theorem 8.14. *For any computable function $f : \mathbb{N} \to \mathcal{L}\,\mathbb{N}$,*

$$\exists (p : \mathbb{N}), \Pi(y : \mathbb{N}), (\{p\}(y) = f(\langle p, y \rangle)) .$$

These follow by functoriality of truncation from the corresponding version for recursive machines.

Similarly, we have the characterization of recursive and recursively enumerable sets.

Definition 8.15. A subset A of $\mathbb{N}$ is *recursively enumerable* when

$$\exists (f : \mathbb{N} \to \mathcal{L}\,\mathbb{N}), \mathsf{isComputable}(f) \times \mathsf{range}(f) = A.$$

Notice that being recursively enumerable is a proposition, and it is the truncation of having recursively enumerable structure.

Theorem 8.16. *A subset A of $\mathbb{N}$ is recursive if and only it is complemented and both A and its complement are recursively enumerable.*

Theorem 8.17. *A subset of $\mathbb{N}$ is r.e. iff it is the domain of a computable function.*

Theorem 8.18. *If A is an inhabited subset of $\mathbb{N}$, then the following are equivalent.*

1. *A is the domain of a computable partial function.*
2. *A is the range of a computable partial function.*
3. *A is the range of a computable total function.*
4. *A is the image of a primitive recursive function.*

8.4 The halting problem

Consider the predicate $H : \mathbb{N} \to \mathbb{N} \to \mathcal{U}$ defined by

$$H(e, x) = \mathsf{defined}(\{e\}(x)).$$

One of the first results in computability theory is that this predicate is not recursive. We repeat the proof here. First, let us define the diagonal function

$$\begin{aligned} d &: \mathbb{N} \to \mathcal{L}\,\mathbb{N} \\ \mathsf{defined}\, d(x) &\stackrel{\mathsf{def}}{=} (\{x\}(x) = \bot), \\ \mathsf{value}(d(x))(p) &\stackrel{\mathsf{def}}{=} 0. \end{aligned}$$

Lemma 8.19. *The function d is not computable.*

Proof. Since we are trying to prove a proposition, we may assume we have a machine m computing d with code e. Then we have

$$\{e\}(e) = \mathsf{eval}(m, e) = d(e).$$

Then we have $\mathsf{defined}\{e\}(e) = \neg(\mathsf{defined}\{e\}(e))$, which is impossible. $\square$

Theorem 8.20 (Undecidability of the halting problem)**.** *The predicate H is not recursive.*

Proof. We wish to derive a contradiction from the assumption $\mathsf{isRec}(H)$. As $\emptyset$ is a proposition, we may assume we have a computable total function $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $f(e, x) = 1$ iff $\{e\}(x)$ is defined and $f(e, x) = 0$ iff $\{e\}(x)$ is not defined.

Then $d(x)$ is pointwise equal to the function

$$d(x) = \begin{cases} 0 & \text{if } f(x, x) = 0 \\ \bot & \text{if } f(x, x) = 1, \end{cases}$$

so that $d(x)$ is defined iff $\neg H(x, x)$ and $d(x)$ is undefined iff $H(x, x)$.

We claim that $d(x)$ is computable. By assumption, f is a total computable function. Moreover, we have that $\lambda x.\bot$ is computable: we know that the constant functions $\lambda x, y.0$ has computable structure, and this is the characteristic function for the relation $\lambda x, y.\mathsf{false}$. Hence, $\lambda x.\mu y.\mathsf{false}$ has computable structure. But as there is no y such that false, we must have that $\mu y.\mathsf{false} = \bot$. So then d is a computable case analysis, and has computable structure. But this contradicts Lemma 8.19. $\square$

8.5 Semidecidable propositions

The Rosolini partial functions provide an abstract notion of semidecidable proposition. A more concrete notion is given by restricting our attention to computable sequences.

Definition 8.21. A *semidecision procedure* for a proposition P is a function $f : 2^{\mathbb{N}}$ with computable structure and at most one 1 such that $P \simeq \langle f \rangle$. That is,

$$\mathsf{SemiDecision}(P) \stackrel{\mathsf{def}}{=} \Sigma(f : 2^{\mathbb{N}}), \mathsf{isProp}\langle f \rangle \times \mathsf{isComputable}(f) \times (P \simeq \langle f \rangle).$$

A proposition P *is semidecidable* when $\mathsf{SemiDecision}(P)$ is inhabited:

$$\mathsf{isSemiDecidable}(P) \stackrel{\mathsf{def}}{=} \|\mathsf{SemiDecision}(P)\|\,.$$

The semidecidable propositions are exactly those which arise as the value of a program. That is,

Theorem 8.22. *For all $A : \mathcal{U}$, there exists a computable partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that $A = f(0)\downarrow$ if and only if A is semidecidable. That is,*

$$\mathsf{isSemiDecidable}(A) \simeq \big(\exists(f : \mathbb{N} \rightharpoonup \mathbb{N}), \mathsf{isComputable}(f) \times (A = f(0)\downarrow)\big).$$

Proof. Suppose we are given a function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ with computability structure t such that such that $A = f(0)\downarrow$. We may define

$$\alpha(n) = \begin{cases} 0 & \text{if } \mathsf{run}_n(t, 0) = \mathsf{inr}\, k \text{ for some } k, \text{ and } \alpha(m) = 1 \text{ for } m < n, \\ 1 & \text{otherwise.} \end{cases}$$

It is easy to see that α is recursive, that $\langle\alpha\rangle$ is a proposition and that $A = \langle\alpha\rangle$.

Conversely, suppose we are given α with recursive structure. Consider the constant function $f(x) = \mu k.(\alpha(k) = 0)$. which also has recursive structure. Moreover, it is clear that $f(0)\downarrow = A$.

The desired equivalence follows by functoriality of truncation. □

Unfortunately, the semidecidable propositions do not form a dominance without additional assumptions (recall Section 5.8, and see Section 9.3), and moreover, the semidecision procedures do not even form a structural dominance (recall Section 5.9). We say more about this in the discussion below, but the key obstacle is the fact that we use a computable function rather than a function with recursive structure. Regardless, we can talk about disciplined maps for the semidecision procedures, but since the semidecision procedures are not closed under Σ, we cannot apply Theorem 5.44 to determine that the *semidecidably-disciplined maps* compose.

8.6 Discussion

Chapter 7 contains the proofs one sees in a classical course on computability theory, while this chapter contains the results. This split is a result of handling structure and property separately. The careful distinction, and the resulting awkward phrasing in Chapter 7 may seem like unnecessary work. The discussion of decidable equality for computable functions in Section 8.2 is the first hint that this distinction is actually meaningful. Chapter 9 provides more evidence that there is technical value in using a framework that allows a distinction between computability as structure and computability as property.

The statement that every partial function $\mathbb{N} \to \mathcal{L}\,\mathbb{N}$ is computable is false, as the diagonal function $d : \mathbb{N} \to \mathcal{L}\,\mathbb{N}$ is not computable. This is the technical reason why we must restrict the available partial functions. If we want to make any sense of the claim "all partial functions are computable", then we must discount partial functions like the above from consideration.

The Rosolini propositions can be seen as an abstract version of the semidecidable propositions, ignoring computability. The restriction to semidecidable propositions not only fails to form a dominance, but does not even give a structural dominance without countable choice. The immediate barrier is that a semidecision procedure comes with a computable function, not a recursive machine. Therefore, we cannot lift the dependent composition used in Lemma 5.41 to the semidecision procedures, unless we use countable choice.

The obvious solution is to use a recursive machine (or a function with recursive structure) in place of a computable function. Indeed, this resolves the issue of choice. However, we will see in Chapter 9, and we saw already in Theorem 8.6 that computable functions and functions with recursive structure are not interchangeable. In particular, the developments in Section 9.3 are more natural with the above notion of semidecision procedure. Since for our purposes the notion of semidecision procedure is an intermediate notion between Rosolini structure and computability, the above notion is more useful. A more focused analysis of semidecidable propositions may find tracking explicit structure in semidecision procedures to be useful.

CHAPTER 9

Computability and partiality

There are three basic threads running through this thesis: restricted notions of partiality; the distinction between structure and property; and the notion of computability. Here we tie these three threads together. Section 9.1 relates partiality and computability by showing that all computable functions must be Rosolini partial functions. Section 9.2 expands on a remark in Chapter 8 concerning the embedding from recursive machines to $\mathbb{N}$, and resolves the two paradoxes set forth in the introduction. Finally, Section 9.3 gives an explicit conjecture concerning which partial functions can be computable, tying together the material of Parts II and III.

9.1 Semidecidable propositions and disciplined maps

We saw in Section 8.4 that there exists non-computable partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$. We spent Chapter 5 foreshadowing this by looking at restrictions of the set of all partial functions. In particular, we examined the set of Rosolini propositions. In computational contexts, the Rosolini propositions correspond to the semidecidable propositions.

Indeed, all computable functions are valued in Rosolini propositions. Recall the relation R_m from Section 7.2 used in the definition of $\mathsf{eval}(m)$.

Theorem 9.1. *For any machine $m : \mathsf{RM}$ and any $x, y : \mathbb{N}$, the types $R_m(x, y)$ and $\Sigma(y : \mathbb{N}), R_m(x, y)$ are Rosolini propositions.*

Proof. For the first type, let $g : \mathbb{N} + \mathbb{N} \to 2$ be the function which takes value 1 on $\mathsf{inr}\, y$ and 0 otherwise. Define $\alpha_k \overset{\mathsf{def}}{=} g(\mathsf{run}_k(m, x))$. We have

$$(\alpha_k = 1) \simeq (g(\mathsf{run}_k(m, x)) = 1);$$

i.e., α takes the value 1 exactly when $\mathsf{run}_k(m,x) = \mathsf{inr}\, y$, so $R_m(x,y) \simeq \exists(k:\mathbb{N}), \alpha_k = 1$. Finally, we truncate α, to get the sequence which takes value 1 only at the first location that α does.

For the second type, let $h : \mathbb{N}+\mathbb{N} \to 2$ be the function which is 0 on $\mathsf{inl}\, k$ and 1 on $\mathsf{inr}\, k$, and again take $\alpha_k = h(\mathsf{run}_k(m,x))$. We have that

$$(\alpha_k = 1) \simeq (\Sigma(y:\mathbb{N}), \mathsf{run}_k(m,x) = \mathsf{inr}\, y).$$

Summing over all $k:\mathbb{N}$, rearranging and truncating takes us to

$$(\exists(k:\mathbb{N}), \alpha_k = 1) \simeq (\exists(y:\mathbb{N}), \Sigma(k:\mathbb{N}), \mathsf{run}_k(m,x) = \mathsf{inr}\, y).$$

By Lemma 2.19, we then have

$$(\exists(k:\mathbb{N}), \alpha_k = 1) \simeq \Sigma(y:\mathbb{N}), \exists(k:\mathbb{N}), \mathsf{run}_k(m,x) = \mathsf{inr}\, y. \qquad \square$$

Corollary 9.2. *If $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is computable, then $f(n)$ is a Rosolini partial element for all $n : \mathbb{N}$.*

Proof. Since we are trying to prove a proposition, we may assume we have some $m : \mathsf{RM}$ computing f. We know that the extent of $f(n)$ is equivalent to $\Sigma(y:\mathbb{N}), R_m(x,y)$, so by Theorem 9.1, $f(n)$ is Rosolini. $\square$

Taking this one step farther, we have

Theorem 9.3. *Every computable partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is disciplined with respect to the Rosolini propositions.*

$$\mathsf{isComputable}(f) \to \mathsf{isDis}(f).$$

Proof. By functoriality of truncation, it is enough to show that for any recursive machine $m : \mathsf{RM}$, there is a function $g : \mathbb{N} \to \mathcal{L}_{\mathsf{RS}}(\mathbb{N})$ such that $\mathsf{tame}(g) = \mathsf{eval}(m)$. This function g is given by $\mathsf{run}^k(m)$. We have the predicate

$$p(n) \overset{\mathsf{def}}{=} \lambda k.k \text{ is least such that } \mathsf{run}^k(n)\downarrow.$$

Then $p(n)$ defines a Rosolini structure, and if $w : (p(n)(k) = 1)$, then $\mathsf{run}^k(n) = \mathsf{inr}\, m_w$ for some $m_w : \mathbb{N}$, and so we can take $g : \mathbb{N} \to \mathcal{L}_{\mathsf{RS}}(\mathbb{N})$ to be the function

$$g(n) \stackrel{\mathsf{def}}{=} \big(p(n), -, \lambda w.m_w\big). \qquad \square$$

In fact, the functions g and h used in the proof of Theorem 9.1 are computable, and so we may strengthen the above results.

Theorem 9.4. *For any machine $m : \mathsf{RM}$ and any $x, y : \mathbb{N}$, the types $R_m(x, y)$ and $\Sigma(y : \mathbb{N}), R_m(x, y)$ are semidecidable propositions.*

Corollary 9.5. *If $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is computable, then f is disciplined with respect to the semidecidable propositions.*

We examine the converse of this last result later in the chapter.

9.2 Which total functions can be computable?

Consider the following two versions of "Church's Thesis":

$$\Pi(f : \mathbb{N} \to \mathbb{N}), \mathsf{isComputable}(\eta \circ f),$$

and

$$\Pi(f : \mathbb{N} \to \mathbb{N}), \mathsf{CompStruct}(\eta \circ f).$$

The first is consistent, with the effective topos as a model. The second, however, is false [84, 7]. Let us call the first *Church's thesis* and the second *strong Church's thesis*. We will follow the structure of Troelstra's proof to show that Strong Church's thesis is false. The proof proceeds as follows:

- If strong Church's thesis holds, then $\mathbb{N} \to \mathbb{N}$ embeds into $\mathbb{N}$, and so has decidable equality.
- if $\mathbb{N} \to \mathbb{N}$ has decidable equality, then there is a function $H : \mathbb{N} \to 2$ deciding the complement of the halting problem.
- if strong Church's thesis holds, then H is computable. Hence, we can recursively decide the complement of the halting problem.

- the complement of the halting problem is not recursively decidable.

We break the proof into lemmas below.

Lemma 9.6. *Assuming strong Church's Thesis, there is an embedding* $\epsilon : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ *such that for any* f,

$$\{\epsilon(f)\} = \eta \circ f.$$

Proof. Given a witness c of Church's thesis we get a $c' : (\mathbb{N} \to \mathbb{N}) \to \mathsf{RM}$ as $c' \stackrel{\mathsf{def}}{=} \pi_0 \circ c$. By definition, we have

$$\mathsf{eval}(c'(f))(n) = (\eta \circ f)(n),$$

so that $\{\mathsf{encode}(c'(f))\} = \eta \circ f$. So we may take

$$\epsilon(f) \stackrel{\mathsf{def}}{=} \mathsf{encode}(c'(f)).$$

Now if $\epsilon(f) = \epsilon(g)$, then

$$\eta \circ f = \{\epsilon(f)\} = \{\epsilon(g)\} = \eta \circ g.$$

Finally, η is an embedding, so $f = g$. □

Corollary 9.7. *If strong Church's thesis holds, then* $\mathbb{N} \to \mathbb{N}$ *has decidable equality. That is, strong Church's thesis implies WLPO.*

Proof. Assuming strong Church's thesis, $(\mathbb{N} \to \mathbb{N})$ embeds into a type with decidable equality. □

Lemma 9.8. *If* $\mathbb{N} \to \mathbb{N}$ *has decidable equality, there is a function* $H : \mathbb{N} \to \mathbb{N}$ *such that* $H(x) = 0$ *iff* $\{x\}(x)$ *is undefined.*

Proof. Fix $x : \mathbb{N}$. Define $r_x : \mathbb{N} \to \mathbb{N}$ as

$$r_x(n) = \begin{cases} 1 & \text{if } \mathsf{run}_n(\{x\}, x) = \mathsf{inr}(y) \text{ for some } y \\ 0 & \text{if } \mathsf{run}_n(\{x\}, x) = \mathsf{inl}(k) \text{ for some } k. \end{cases}$$

That is, $r_x(n) = 1$ iff $\{x\}(x)$ returns in under n steps.

By our assumption that $\mathbb{N} \to \mathbb{N}$ has decidable equality, we have

$$(r_x = \lambda k.0) + \neg(r_x = \lambda k.0)$$

Note that $r_x = \lambda k.0 \Leftrightarrow \{x\}(x) = \bot$.

Define $H(x) : \mathbb{N} \to \mathbb{N}$ by

$$H(x) = \begin{cases} 1 & \text{if } r_x \neq \lambda n.0 \\ 0 & \text{if } r_x = \lambda n.0. \end{cases}$$

We have

$$H(x) = 0 \Leftrightarrow r_x = \lambda k.0 \Leftrightarrow \{x\}(x) = \bot. \qquad \square$$

Theorem 9.9 (Troelstra [84])**.** *Strong Church's thesis is false.*

Proof. If strong Church's Thesis holds, by Lemma 9.7 we have that $\mathbb{N} \to \mathbb{N}$ has decidable equality, and so by Lemma 9.8 we have a computable $H : \mathbb{N} \to \mathbb{N}$ such that $H(x) = 0$ iff $\{x\}(x)$ is undefined. But we saw in Section 8.4 that this function cannot be computable. $\square$

As any reasonable notion of partial function includes all total functions, we cannot have a version of strong Church's thesis for partial functions. Then in particular, if d is any set of propositions containing $\mathbf{1}$, we have

$$\neg\Pi(f : \mathbb{N} \to \mathcal{L}_{\mathsf{d}}(\mathbb{N})), \mathsf{CompStruct}(f).$$

However, the weaker form $\Pi(f : \mathbb{N} \to \mathcal{L}_{\mathsf{d}}(\mathbb{N})), \mathsf{isComputable}(f)$ is consistent for $\mathsf{d} = \mathsf{isContr}$. The above proof hinges on the fact that there is an embedding from the type of functions with recursive structure to $\mathbb{N}$. In the above, we extend this to get an embedding from $(\mathbb{N} \to \mathbb{N})$ to $\mathbb{N}$. Recall that in Section 8.2 we argued that we cannot expect an embedding from the type of all computable functions to $\mathbb{N}$. Indeed, while this is true classically, it is contradicted by Church's thesis.

Theorem 9.10. *If Church's thesis holds, then there is no embedding*

$$\big(\Sigma(f : \mathbb{N} \rightharpoonup \mathbb{N}), \mathsf{isComputable}(f)\big) \to \mathbb{N}.$$

Proof. We essentially follow the argument for Theorem 9.9. Church's thesis gives us an embedding $(\mathbb{N} \to \mathbb{N}) \to \Sigma(f : \mathbb{N} \rightharpoonup \mathbb{N}), \mathsf{isComputable}(f)$. Extending this along the hypothetical embedding e gives an embedding $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$. As a subtype of a type with decidable equality, $\mathbb{N} \to \mathbb{N}$ has decidable equality. Then, by Lemma 9.8, we have a function $H : \mathbb{N} \to \mathbb{N}$ deciding the halting problem. By Church's thesis, H is computable, which is impossible. $\square$

In the introduction, we mentioned four facts about computability theory arising from topos theoretic and type theoretic intuition.

C1 It is *consistent* that all total functions $\mathbb{N} \to \mathbb{N}$ are computable.

C2 It is *false* that all total functions $\mathbb{N} \to \mathbb{N}$ are computable.

C3 There is an embedding from the class of computable functions to the natural numbers.

C4 It cannot be proved that there is an embedding from the class of computable functions to the natural numbers.

C2 is Theorem 9.9, while C3 is an easy consequence of Theorem 7.14. C1 and C4 correspond to the incompatible principles in Theorem 9.10. We know that Church's thesis holds in the effective topos, and so is consistent with pure MLTT (but see the discussion below about univalent mathematics), and so we have strong reason to expect C1 and C4 to hold. The conflict between the topos-theoretic and type-theoretic facts has evaporated: C2 and C3 refer to computability as structure, while C1 and C4 refer to computability as property.

This is not an artefact of univalent mathematics: Indeed, Troelstra's argument was originally used to show that function extensionality, Church's thesis and the axiom of choice are mutually incompatible over HA^ω. Moreover, in the effective topos, with the Mitchell-Benabou language, all total functions are computable, and there is no embedding from the class of computable functions to $\mathbb{N}$, but in the effective topos as a model of MLTT, it is false that all total functions are computable, and there is an embedding from the class of computable functions to $\mathbb{N}$.

9.3 Which partial functions can be computable

We have spent several chapters now circling around claims about which partial functions can be computable. It is time to approach the question directly. We have so far presented two sets of propositions: the Rosolini propositions, which for this section we call R, and the semidecidable propositions which for this section we call S. For each of these, two notions of partial function: the R- and S-partial functions, and the R- and S-disciplined maps. One may expect that the S-disciplined maps are exactly the computable functions—that the converse of Corollary 9.5 holds, but it seems that this result needs both countable choice and Church's thesis to prove. Before presenting the result explicitly (Theorem 9.14), let's examine the effect of Church's thesis on Rosolini structures.

Theorem 9.11. *Church's thesis implies that semidecision functions and Rosolini structures coincide. That is, assuming Church's thesis we have*

$$\Pi(P : \mathcal{U}), \mathsf{SemiDecision}(P) \simeq \mathsf{rosoliniStructure}(P)$$

Proof. We have that

$$\mathsf{SemiDecision}(P) \overset{\mathsf{def}}{=} \Sigma(f : 2^{\mathbb{N}}), \mathsf{isProp}\langle f\rangle \times \mathsf{isComputable}(f) \times (P \simeq \langle f\rangle).$$

By Church's thesis, we have that $\mathsf{isComputable}(f)$ is true for all $f : \mathbb{N} \to 2$, and so we have

$$\mathsf{SemiDecision}(P) \simeq \Sigma(f : 2^{\mathbb{N}}), \mathsf{isProp}\langle f\rangle \times (P \simeq \langle f\rangle),$$

and the latter is the definition of Rosolini structures on P. $\square$

That is, Church's thesis collapses R and S. So we have

Corollary 9.12. *Assuming Church's thesis,*

(a) Rosolini propositions and semidecidable propositions coincide;

(b) The Rosolini structure lifting and the semidecision function lifting coincide;

(c) Rosolini partial functions and semidecidable partial functions coincide;

(d) Rosolini-disciplined maps and semidecidable-disciplined maps coincide.

Recall Theorem 5.46, which says that under countable choice, the disciplined maps coincide with the partial functions $\mathbb{N} \to \mathcal{L}_{\mathsf{isRosolini}}\,\mathbb{N}$.

Corollary 9.13. *Assuming both countable choice and Church's thesis, the following all coincide*

(a) the Rosolini partial functions, $\mathbb{N} \to \mathcal{L}_{\mathsf{R}}\,\mathbb{N}$*;*

(b) the Rosolini-disciplined maps, $\mathsf{Dis}_{\mathsf{R}}(\mathbb{N}, \mathbb{N})$*;*

(c) the semidecidable partial functions, $\mathbb{N} \to \mathcal{L}_{\mathsf{S}}\,\mathbb{N}$*;*

(d) the semidecidable-disciplined maps, $\mathsf{Dis}_{\mathsf{S}}(\mathbb{N}, \mathbb{N})$*;*

In short, Church's thesis and countable choice together unite all the restricted notions of partial function we have considered so far. There is one notion still missing: the computable partial functions. Indeed, countable choice and Church's thesis together imply that the Rosolini and computable partial functions coincide.

Theorem 9.14. *Assuming countable choice and Church's thesis, any function* $\mathbb{N} \rightharpoonup \mathbb{N}$ *which is valued in Rosolini propositions is computable.*

Proof. By Corollary 9.13, it is enough to show (under the two assumptions) that any semidecidable-disciplined map is computable. Since we are trying to prove a proposition, we may assume we are given an explicit $f : \mathbb{N} \to \mathcal{L}_{\mathsf{S}}(\mathbb{N})$, and show that $\mathsf{tame}(f)$ is computable.

Decomposing f and applying Theorem 8.22, we have $F : \mathbb{N} \to \mathcal{U}$ such that

$$\Pi(n : \mathbb{N}), \Big(\exists(g : \mathbb{N} \to \mathbb{N}), \mathsf{isComputable}(g) \times (F(n)\downarrow = g(0)\downarrow)\Big) \times \big(F(n) \to \mathbb{N}\big).$$

By countable choice, then, we have

$$\Big\|\Pi(n : \mathbb{N}), \Big(\Sigma(g : \mathbb{N} \to \mathbb{N}), \mathsf{CompStruct}(g) \times (F(n)\downarrow = g(0)\downarrow)\Big) \times \big(F(n) \to \mathbb{N}\big)\Big\|.$$

Again, we are trying to prove a proposition, so we may assume we are given some H of type

$$H : \Pi(n : \mathbb{N}), \Big(\Sigma(g : \mathbb{N} \to \mathbb{N}), \mathsf{CompStruct}(g) \times (F(n)\downarrow = g(0)\downarrow)\Big) \times \big(F(n) \to \mathbb{N}\big).$$

We can again reorganize our type, to get a function $M : \mathbb{N} \to \mathbb{N}$ such that

$$\Pi(n : \mathbb{N}), F(n)\downarrow = \{M(n)\}(0)\downarrow \times F(n) \to \mathbb{N}$$

Now we can consider $h : \mathbb{N} \rightharpoonup \mathbb{N}$ whose extent is given by $F(n)$ and whose value is given by the map $f_n : F(n) \to \mathbb{N}$ which is the last component of $H(n)$. By definition, this is pointwise equal to $\mathsf{tame}(f)$, so we need a recursive machine computing h. For each $n : \mathbb{N}$, let m_n be the machine for g where $F(n)\downarrow = g(0)\downarrow$. Define the machine m which acts as follows:

To initialize m on input n, initialize m_n on input 0, and ensure we are in state n. In state n, to transition, use the transition function of m_n and stay in state n. If we have halted in state n, then we know $g(0)\downarrow$, and so we have $w : F(n)\downarrow$, and we can return $f_n(w)$.

The only question is whether we can computably access the machines m_n, but each m_n is coded by $M(n)$, and by Church's thesis M is computable. $\square$

This gives us an extension of Corollary 9.12.

Theorem 9.15. *Assuming both countable choice and Church's thesis, the following all coincide*

(a) the Rosolini partial functions, $\mathbb{N} \to \mathcal{L}_{\mathsf{R}}\,\mathbb{N}$*;*
(b) the Rosolini-disciplined maps, $\mathsf{Dis}_{\mathsf{R}}(\mathbb{N}, \mathbb{N})$*;*
(c) the semidecidable partial functions, $\mathbb{N} \to \mathcal{L}_{\mathsf{S}}\,\mathbb{N}$*;*
(d) the semidecidable-disciplined maps, $\mathsf{Dis}_{\mathsf{S}}(\mathbb{N}, \mathbb{N})$*;*
(e) the computable partial function $\Sigma(f : \mathbb{N} \rightharpoonup \mathbb{N}), \mathsf{isComputable}(f)$*.*

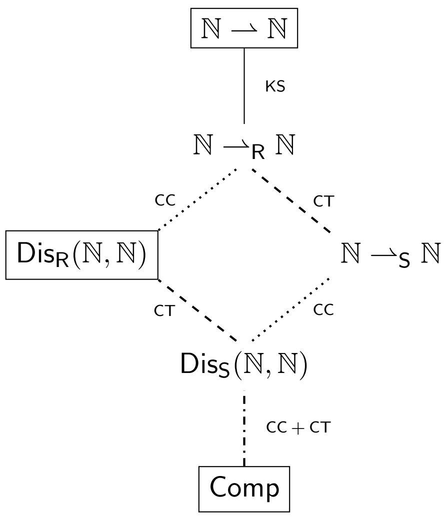
9.4 Discussion

We know that countable choice and Church's thesis hold in the effective topos. While the effective topos satisfies proposition extensionality, propositions are handled differently in univalent mathematics than in topos logic. Nevertheless, we have reason to believe the following

Conjecture. *It is consistent that all Rosolini-disciplined maps are computable.*

To prove this, it is enough to give a model of MLTT with proposition and function extensionality in the univalent style which validates countable choice and Church's thesis. While such a model would actually give the stronger result that it is consistent that Rosolini partial functions are computable, we have intentionally chosen to state the weaker conjecture. The Rosolini-disciplined maps can be shown to compose even in the absence of choice principles, and so can be used as a surrogate version of the computable partial functions, even in settings where we do not have countable choice, so long as such settings are compatible with Church's thesis.

The distinction between the stronger and weaker form of the above conjecture speak towards our goal of understanding how the computable functions fit into a constructive framework. Our results on this point can be summarized by looking at possible ways of collapsing the following diagram:



Lower types embed into higher types and the boxed types can be shown to be closed under composition with no choice principles; the dashed lines collapse under Church's thesis, while the dotted lines collapse under countable choice. Moreover, the solid line collapses under Kripke's schema.

Since we know not all partial functions are computable, a corollary of Theorem 9.14 is that countable Choice, Church's thesis and Kripke's schema are incompatible. The stronger result that Kripke's Schema and Church's thesis are incompatible is already known, but the proof in [85] goes via a different route.

The argument used in Theorem 9.9 was first given to show that function extensionality, Church's thesis and the axiom of choice are mutually inconsistent over HA^{ω}. Note that under choice, Church's thesis is the truncation of strong Church's thesis.

Further work

There are two clear lines of research from here: The first is to resolve the model-theoretic aspects in order to prove the conjecture, and the second is to develop constructive computability theory at higher types.

A model validating Church's thesis

A naive sketch of the proof goes as follows: We've already shown that Church's Thesis and Countable choice imply that the computable partial functions are exactly the Rosolini disciplined functions. Moreover, we know that the effective topos models both countable choice and Church's Thesis, and so in the effective topos the Rosolini disciplined maps coincide with the partial computable functions.

Unfortunately, the notion of proposition as element of the subobject classifier, and the notion of proposition as subsingleton type do not quite line up: in fact, it is not the case that all subsingletons of the effective topos are equal. In other words, proposition extensionality as stated in Definition 2.28 is not meaningful in the effective topos. There seem to be two ideas one can use to get around this: one is essentially semantic, and the other syntactic.

On the semantic side, some work has been done By Ian Orton and Andy Pitts [63] on using partial elements to give a version of the interval and the Kan filling operation used in cubical type theory. While we don't expect arbitrary toposes to satisfy the univalence axiom (or even proposition extensionality), these ideas could be used to rework the effective topos into a univalent form. Work has already been done on a version of *cubical realizability*, in work by Harper and others on computational higher type theory [3, 15]; ideas from here, or the stack models [26] will likely be of value.

On the syntactic side, it may be possible to interpret our type theory in one that is modeled by the effective topos. Since we do not use full univalence in the above, and function extensionality is true in the effective topos, the sticking point is proposition extensionality. However, since propositions in a topos are subobjects of the (specified) terminal object, they do satisfy proposition extensionality. That is, the only obstacle is that the notions of proposition do not coincide. It should be possible to give an intermediate type theory with a type Prop of propositions, along with a decomposition of the truncation operator $\|-\| : \mathcal{U} \to \mathcal{U}$ through this. It is known that truncation forms an idempotent monad [86, 72], so we could add to a type theory such as CiC rules corresponding to operators $|-| : \mathcal{U} \to \mathsf{Prop}$ and $\mathsf{Elem} : \mathsf{Prop} \to \mathcal{U}$ decomposing this monad as an adjunction. In particular, if Prop is extensional (that is, $P \leftrightarrow Q$ then $P = Q$ for $P, Q : \mathsf{Prop}$), then Elem would pick a canonical subsingleton for each equivalence class of subsingletons. The difficulties in this approach arise in two places:

1. First, some care would be required to ensure we correctly capture the relationship between types that truncation represents. For example, when $P(x)$ is a subsingleton for all $x : X$ we want $\mathsf{Elem}(\forall(x : X), |P(x)|)$ to be equivalent to $\Pi(x : X), P(x)$. Effectively characterizing such properties may be non-trivial.
2. Second, we need some sort of conservativity result, but it's not clear how to state such a conservativity result for this situation.

There's an additional benefit of the second approach: For mathematicians working in a univalent setting, there's a general impression that the univalent approach restricted to the level of sets and propositions is ultimately little more than a more convenient internal language for a topos. A type theory bridging the gap between univalent type theory and a type theory modeled by general toposes would give technical support to this impression.

Higher-type computation

Despite the close ties between the development of constructive theories and computation at higher types, there does not seem to be a coherent constructive development of higher-type computability theory.

The starting point of higher-order computability is PCF, and indeed it is not difficult to give the operational semantics for PCF constructively: Recall that PCF is the simply type lambda calculus with base type ι (representing the naturals, with a zero-test ifz and predecessor pred) and a fixed point combinator $\mathbf{Y}_\sigma : (\sigma \to \sigma) \to \sigma$ for each type σ. Specifically, we have the term grammar

$$\begin{aligned} M ::= \ & x \mid \lambda x.M \mid M(M) \mid \mathbf{Y}_\sigma(M) \mid \\ & \overline{m} \mid \mathsf{succ}(M) \mid \mathsf{pred}(M) \mid \mathsf{ifz}(M, M, M) \end{aligned}$$

with the expected typing rules where ifz is first-order. Note that we take the constants to be constructors rather than combinators. We have

$$\overline{n} \Downarrow \overline{n} \qquad \lambda x.M \Downarrow \lambda x.M \qquad x \Downarrow x$$

$$\frac{M \Downarrow \overline{n}}{\mathsf{succ}\, M \Downarrow \overline{n+1}}$$

$$\frac{M \Downarrow \overline{0}}{\mathsf{pred}\, M \Downarrow \overline{0}} \qquad \frac{M \Downarrow \overline{n+1}}{\mathsf{pred}\, M \Downarrow \overline{n}}$$

$$\frac{L \Downarrow \overline{0} \qquad M \Downarrow v}{\mathsf{ifz}\, LMN \Downarrow v} \qquad \frac{L \Downarrow \overline{n+1} \qquad N \Downarrow v}{\mathsf{ifz}\, LMN \Downarrow v}$$

$$\frac{M(\mathbf{Y}M) \Downarrow v}{\mathbf{Y}M \Downarrow v}$$

$$\frac{L \Downarrow \lambda x.N \qquad N[M/X] \Downarrow v}{LM \Downarrow v}$$

The Scott model has interpretation at base type $[\![\iota]\!] = \mathbb{N}_\perp$, the dcpo of the lifted naturals. Constructively, we may replace this with our lifting. Note that we cannot constructively use the poset given by adding an element $\perp$ below every element of $\mathbb{N}$ (with the discrete ordering), since this type $\mathbb{N}_\perp$ can not be shown to be a dcpo constructively. In fact, the two-element poset with $0 \leq 1$ is a dcpo precisely if excluded middle holds: fix a proposition P and consider the family $u : P + 1 \to 2$ defined by $u(\mathsf{inl}\, p) = 1$ and $u(\mathsf{inr}\, \star) = 0$. Then this is directed as the only possible non-equal pairs of elements are $\mathsf{inl}\, p$ and $\mathsf{inr}\, \star$, for some $p : P$. In this case, we have $u(\mathsf{inr}\, \star) < u(\mathsf{inl}\, p)$. Then if the supremum of u is 1, then p holds, and if it is 0, then p does not hold, but as $u : 2$ one of these cases must hold.

To see an example of how a constructive development of higher-type computability differs from the classical development, we sketch an example based on the metric model of PCF in [28]. There, Escardó presents a model of PCF in complete ultrametric spaces where we use the metric to count recursive unfoldings—two points are closer if we need to unfold more applications of the fixed-point combinator to distinguish them. We can define a logical relation (in fact, a partial equivalence relation at each type) of extensional equality between elements of this model, and then we can use a logical relation between the metric model and the Scott model to show that the Scott model is a subquotient (the *extensional collapse*) of the ultrametric model.

The situation is somehow reversed in the classical case, since we must use $\mathcal{L}(\mathbb{N})$ instead of $\mathbb{N}_\perp$ to interpret the base type.

PCF and ultrametric spaces

An *ultrametric space* M is a metric space where the distance function d satisfies the *strong triangle inequality*,

$$d(x, z) \leq \mathsf{max}\{d(x, y), d(y, z)\}.$$

Real numbers are not essential in the development. We instead avoid the technical issues that arise from working with the reals by presenting ultrametric spaces via a sequence of equivalence relations.

Definition 9.16. An *ultrametric space* is a set M equipped with countably many equivalence relations $\{=_n\}_{n:\mathbb{N}}$ such that

1. $=_n$ is an equivalence relation for all n;
2. $=_0$ is the total relation;
3. if $x =_{n+1} y$, then $x =_n y$;
4. if $x =_n y$ for all n, then $x = y$.

The computational intuition is that $x =_n y$ when it takes at least n steps to distinguish x and y. By setting $d(x, y) = \mathsf{inf}\{2^{-k} \mid x =_k y\}$, we get an ultrametric space in the classical sense from an ultrametric space as above.

We can define Cauchy sequences, limits and Cauchy completeness in the expected way, and take a function between ultrametric spaces to to be *non-expansive* if $x =_n y \Rightarrow f(x) =_n f(y)$, and *contractive* if $x =_n y \Rightarrow f(x) =_{n+1} f(y)$.

Then we can prove that the category of complete ultrametric spaces with non-expansive maps is cartesian closed, and take $X \to Y$ be the (complete) ultrametric space of non-expansive maps. We can also show that $\mathsf{D}(X)$ is a complete ultrametric space for any type X, and give a version of the Banach fixed-point theorem. From this, we can model PCF in complete ultrametric spaces by taking $\mathsf{D}(\mathbb{N})$ to interpret the base type, and using the fixed-point theorem to interpret fixed-point recursion. Let us use D to denote this model. We can also give an operational semantics which tracks recursive unfoldings, and show that D is adequate with respect to this semantics.

The logical relation of *extensional equality* on D is given by saying that $\mathsf{delay}^k(n) \approx \mathsf{delay}^l(n)$ for each $k, l, n : \mathbb{N}$, and then lifting this to higher types in the usual way. The quotient of the base type D_ι by this relation is $\mathcal{L}_{\mathsf{isRosolini}}(\mathbb{N})$. Classically, this is equivalent to $\mathcal{L}(\mathbb{N})$, and this quotient lifts to higher types. Constructively, however, we do not have this equivalence. Worse, we do not know that $\mathcal{L}_{\mathsf{isRosolini}}(\mathbb{N})$ is a dcpo, so we cannot use a version of the Scott model replacing $\mathcal{L}(\mathbb{N})$ with $\mathcal{L}_{\mathsf{isRosolini}}(\mathbb{N})$. In other words, the interpretation of a type σ in the Scott model is not a subquotient of D_σ.

How much, then, can we say constructively about the relationship between the ultrametric and Scott models?

Conclusion and summary

Here we summarize the material in the chapter discussions: a key point in the univalent perspective is the careful distinction between structure and property, while taking the former to subsume the latter. There is a distinction between property and structure in logical systems (with propositions as formulas), and some type theories (such as the calculus of inductive constructions [21, 25, 41], which has a type of propositions), but there is usually no syntax for relating structure and property. It is worth pointing out that propositions are not exactly *proof-irrelevant* in our framework (as suggested in work leading up to the notion such as [65, 60, 64, 35]), because information can be extracted from them (Lemma 3.13 and [48, 49, 47]).

The difference in treatment of propositions makes direct translation of known results (especially model-theoretic results) somewhat non-trivial. Internally, a choice between structure and property is necessary at all points (but the correct choice often presents itself quickly); externally, we have difficulty using many standard models of MLTT, since they do not satisfy our version of proposition extensionality. On the other hand, this treatment of propositions resolves some ambiguity surrounding choice and extensionality (consider Section 3.6 and [58].

Besides the treatment of propositions, there are 3 novelties of univalent mathematics

- The namesake due to Voevodsky, the *univalence axiom* which gives a general extensionality principle, from which proposition and function extensionality follow.
- The stratification of types into levels determined by the structure of identity types—contractible types; propositions whose identity types are contractible; sets whose identity types are propositions, (1-)groupoids whose identity types are sets, 2-groupoids whose identity types are groupoids, and so on.
- Higher-inductive types (HITs), which allow us to generate a type using not only constructors of that type, but also *path constructors* which inhabit identity types.

The status of HITs and the relationship between univalence and computational meaning are active areas of investigation [54, 20, 27, 45, 15, 46, 40, 8, 18, 39]. In the above, we restricted most of our attention to sets, limiting our use of these new notions to quotient types, (propositional) truncations, proposition extensionality, and function extensionality.

Our approach to partiality builds off of past approaches to partiality in type theory and in topos theory. On the type-theoretic side, there are versions of the *delay monad* [12, 14]. At its most basic, the delay monad is an intensional approach to partial functions with a clock. Attempts have been made to make it more extensional by quotienting [16], but this approach requires countable choice for composability; or generating it as a higher-inductive type [2], but this approach expands the type of Rosolini partial functions, where we want to restrict the Rosolini partial functions: we are trying to represent the computable partial functions, and this type embed into the Rosolini partial functions. On the topos-theoretic side, there is the notion of dominance [75] which is used to restrict the type of partial elements. Of particular interest is the set of *Rosolini propositions*, which is used as a form of semidecidable propositions in synthetic domain theory [75, 44, 89, 68] and synthetic computability theory [6].

Several traditional taboos can be stated in terms of Rosolini propositions:

- Kripke's Schema says that all propositions are Rosolini.
- Markov's Principle is double-negation elimination for Rosolini propositions.
- LPO says that true and false are the only Rosolini propositions.
- WLPO says that Rosolini propositions are either false or not false.

However, in order to show that the Rosolini propositions form a dominance, some version of countable choice is needed. In fact, we need exactly that we can recover Rosolini structures from Rosolini propositions (Theorem 5.32). In order for the partial functions arising from a set of propositions to be composable, they must form a dominance. Since partial functions ought to compose, we need countable choice to see the Rosolini partial functions as a good notion of partial function.

Instead, we consider *disciplined maps*—those partial functions which respect information from an intensional version of the Rosolini dominance. This intensional Rosolini dominance (the *Rosolini structures*) gives a lifting which is equivalent to the delay monad, so we can say that a disciplined map from X to Y is a partial function $f : X \rightharpoonup Y$ for which there exists (as a proposition) a Kleisli function for the delay monad which f factors through.

Since countable choice holds in the setoid model, and does not in univalent type theory [26], it is worth examining how things play out in the setoid approach. We can indeed construct the delay monad, and then quotient it to arrive at the Rosolini lifting. In general, the setoid approach has the following advantages over the univalent approach:

1. The setoid approach allows us to use extensionality principles in a simpler version of MLTT.
2. countable choice holds in the setoid model.
3. The setoid approach fits squarely with the traditional account of constructive mathematics proposed by Bishop.

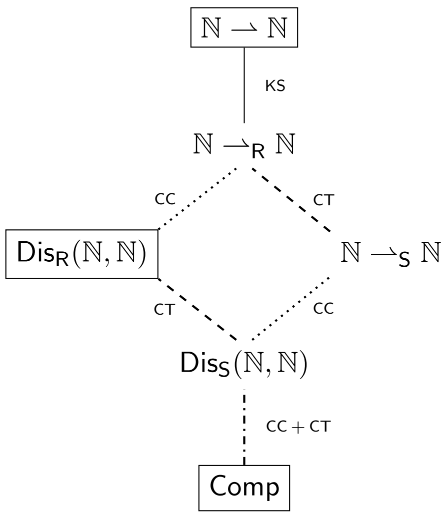
However, the univalent approach has the following advantages over the setoid approach

1. The univalent approach allows us to make use of universes; in particular, this allows us to define more computable functions, and to develop computability in more interesting domains.
2. Definitions are more uniform, since we use an inductively defined identity type and all operations are automatically extensional.
3. The univalent perspective fits squarely with a structural account of mathematics, and extends the traditional account of constructive mathematics.
4. A type of propositions fits naturally into the theory, and clarifies the relationship between structure and property.
5. There are computer proof systems with native support for formalization in univalent styles. No such support exists for the setoid approach.

Once we have a notion of partial function, we can turn to computability theory. Much of the classical theory goes through with little modification, however we introduced a modest abstraction from Turing machines (the *recursive machines*) to simplify notation. We developed computability theory in two forms: with computability as structure ($\mathsf{CompStruct}(f)$; Chapter 7), and with computability as property ($\mathsf{isComputable}(f)$; Chapter 8). Many foundational results in a first course on recursion theory go through for either notion, but a few (such as the Normal Form Theorem) can only be stated in terms of computability as structure.

Although every definable total function is computable, the statement that every partial function $\mathbb{N} \to \mathcal{L}\,\mathbb{N}$ is computable is false, as the diagonal function $d : \mathbb{N} \to \mathcal{L}\,\mathbb{N}$ is not computable. As one goal is to find a good notion of partial function (in particular, we should be able to compose them) which can consistently be assumed to be the computable partial functions, we need to restrict the set of all partial functions in some way.

The Rosolini propositions give the ideas for a first attempt, and we can restrict these to those which can be witnessed to be Rosolini by a computable function to arrive at the *semidecidable propositions*. In any case, neither the Rosolini partial functions nor the semidecidable partial functions are composable, so we must pass instead to the associated disciplined maps. And in fact, countable choice and Church's thesis together imply that the four notions (Rosolini and semidecidable partial functions, Rosolini disciplined maps and Semidecidable disciplined maps) coincide, and in fact are exactly the computable functions. We summarize this in the following diagram:



Lower types embed into higher types and the boxed types can be shown to be closed under composition with no choice principles; the dashed lines collapse under Church's thesis, while the dotted lines collapse under countable choice. Moreover, the solid line collapses under Kripke's schema.

We know that countable choice and Church's thesis hold in the effective topos. Although the handling of propositions in a topos is slightly different than in univalent mathematics, we therefore have reason to believe the following

Conjecture. *It is consistent that all Rosolini-disciplined maps are computable.*

While the obvious model validating this (some univalent version of the effective topos) should also validate the logically stronger statement that all Rosolini partial functions are computable, the Rosolini-disciplined maps can be shown to compose even in the absence of choice principles, and so can be used as a surrogate version of the computable partial functions, even in settings where we do not have countable choice, so long as such settings are compatible with Church's thesis. The fact that the effective topos does not model our version of proposition extensionality leaves us with a direction for future work: find either a way to interpret univalent type theory in the effective topos (or toposes more generally), or come up with a univalent version of the effective topos.

As our primary aim was not to develop computability theory, but to fit it into a larger context, we have not done a great deal of computability theory. This presents another area for future work. As higher-typed computability seems particularly well-suited to study from a constructive view-point (relying on many of the same tools), and there are non-trivial questions raised almost immediately when we begin the attempt (for example, what does the Scott model look like constructively?), this seems to be a fruitful area of study.

An important point in the above development is that we can view a notion as either structure or as property, and the notion will behave differently depending on the view we take. We illustrated this with Church's thesis (which is false for computability as structure), and with our "facts" C1-C4 from the introduction and Section 9.2. We end by emphasizing this point with another anecdote about the development of Part III: Originally, I had stated Theorem 8.22 in a stronger form, about semidecision procedures, giving the weaker statement as a corollary. The proof looked the same, except truncation was applied differently. In fact, the proof was wrong, because I used truncations incorrectly. If I had untruncated all structure in the theorem (using computability structure everywhere) the corresponding result would hold using the same proof, but Theorem 9.11 would not hold, and Corollary 9.12 would need to be modified accordingly. In short, each decision about whether to view computability as property or structure leads to a different outcome. The development of ideas in Sections 8.5 and 9.2 is highly dependent on proper handling of the distinction between structure and property.

Index of Symbols

Named operators such as isProp point to the corresponding entry in the main index.

Index

References

[1] Peter Aczel. The strength of Martin-Löf's intuitionistic type theory with one universe. In Miettinen Seppo and Vänänen Jouko, editors, *Proceedings of the symposiums on mathematical logic in Oulu 1974 and in Helsinki 1975*, 1977. Report No. 2.

[2] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai. Kraus. Partiality, revisited: The partiality monad as a quotient inductive-inductive type. In *FoSSaCS Proceedings*, 2017.

[3] Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher type theory I: abstract cubical realizability. *CoRR*, abs/1604.08873, 2016.

[4] Steven Awodey and Andrej Bauer. Propositions as [types]. *J. Log. and Comput.*, 14(4):447–471, August 2004.

[5] Andrej Bauer. *The realizability approach to computable analysis and topology*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, USA, 2000.

[6] Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5 – 31, 2006. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

[7] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer Verlag, 1980.

[8] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, 2014.

[9] Errett Bishop. *Foundations of constructive analysis*. McGraw-Hill series in higher mathematics. McGraw-Hill, 1967.

[10] Errett Bishop. Mathematics as a numerical language. In A. Kino, J. Myhill, and R.E. Vesley, editors, *Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo N.Y. 1968*, volume 60 of *Studies in Logic and the Foundations of Mathematics*, pages 53 – 71. Elsevier, 1970.

[11] Errett Bishop. *Schizophrenia in Contemporary Mathematics*. American Mathematical Society, 1973.

[12] Ana Bove and Venanzio Capretta. *A Type of Partial Recursive Functions*, pages 102–117. Springer, Berlin, Heidelberg, 2008.

[13] Douglas Bridges and Fred Richman. *Varieties of constructive mathematics*, volume 97 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1987.

[14] Venancio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 15:1–28, 2005.

[15] E. Cavallo and R. Harper. Computational Higher Type Theory IV: Inductive Types. *ArXiv e-prints*, January 2018.

[16] James Chapman, Tarmo Uustalu, and Niccolò Veltri. *Quotienting the Delay Monad by Weak Bisimilarity*, pages 110–125. Springer, 2015.

[17] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory. `https://agda.readthedocs.io/en/latest/language/cubical.html`.

[18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[19] ADT Coq. The coq proof assistant reference manual, 2009.

[20] T. Coquand, S. Huber, and A. Mörtberg. On Higher Inductive Types in Cubical Type Theory. *ArXiv e-prints*, February 2018.

[21] T. Coquand and Gérard Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[22] T. Coquand, B. Mannaa, and F. Ruch. Stack semantics of type theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–11, June 2017.

[23] Thierry Coquand. Some contributions of Vladimir Voevodsky to dependent type theory. Presentation at EUTypes 2018 Working Meeting, 2018.

[24] Thierry Coquand and Nils Anders Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105–1120, 2013.

[25] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[26] Thierry Coquand, Bassel Mannaa, and Fabian Ruch. Stack semantics of type theory. In *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on*, pages 1–11. IEEE, 2017.

[27] Peter Dybjer and Hugo Moeneclaey. Finitary higher inductive types in the groupoid model. In *MFPS XXXIII*, Electronic Notes in Theoretical Computer Science, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. 33rd Conference on Mathematical Foundations of Programming Semantics (to appear).

[28] Martín Escardó. A metric model of PCF, 1998.

[29] Martín Escardó et al. Typetopology. Available at `https://github.com/martinescardo/TypeTopology`.

[30] Martín H. Escardó. Infinite sets that satisfy the principle of omniscience in any variety of constructive mathematics. *The Journal of Symbolic Logic*, 78(3):764–784, 2013.

[31] Martín Hötzel Escardó and Cory M. Knapp. Partial elements and recursion via dominances in univalent type theory. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 82. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[32] Solomon Feferman. Iterated inductive fixed-point theories: Application to hancock's conjecture. In George Metakides, editor, *PATRAS LOGIC SYMPOSION*, volume 109 of *Studies in Logic and the Foundations of Mathematics*, pages 171 – 196. Elsevier, 1982.

[33] Nicola Gambino, Notes by C. Kapulkin, and P. L. Lumsdaine. The univalence axiom and functional extensionality. Lecture notes, 2011.

[34] Edward Griffor and Michael Rathjen. The strength of some Martin-Löf type theories. *Archive for Mathematical Logic*, 33(5):347–385, Oct 1994.

[35] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[36] Martin Hofmann. On the interpretation of type theory in locally cartesian closed categories. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic*, pages 427–441, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[37] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

[38] Martín Hötzel Escardó and Chuangjie Xu. The inconsistency of a brouwerian continuity principle with the curry–howard interpretation. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[39] S. Huber. Canonicity for Cubical Type Theory. *ArXiv e-prints*, July 2016.

[40] Simon Huber. *Cubical interpretations of type theory*. PhD thesis, University of Gothenburg, 2016.

[41] Gérard Huet. Induction principles formalized in the calculus of constructions. In Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT '87*, pages 276–286, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[42] John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000.

[43] J. M. E. Hyland. The effective topos. *Studies in Logic and the Foundations of Mathematics*, 110:165–216, 1982.

[44] J. M. E. Hyland. *First steps in synthetic domain theory*, pages 131–156. Springer, 1991.

[45] Ambrus Kaposi and András Kovaács. A syntax for higher-inductive inductive types, Feb 2018. Submitted to FSCD.

[46] C. Kapulkin and P. L. Lumsdaine. The Simplicial Model of Univalent Foundations (after Voevodsky). *ArXiv e-prints*, November 2012.

[47] N. Kraus, M.H. Escardó, T. Coquand, and T. Altenkirch. Notions of anonymous existence in Martin-Löf type theory. *Logical Methods in Computer Science*, 2016.

[48] Nicolai Kraus. The truncation map $|\_| : \mathbb{N} \to \|\mathbb{N}\|$ is nearly invertible. `https://homotopytypetheory.org/2013/10/28/the-truncation-map-_-%E2%84%95-%E2%80%96%E2%84%95%E2%80%96-is-nearly-invertible/`, 2013.

[49] Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Generalizations of Hedberg's theorem. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications*, pages 173–188, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[50] James Ladyman and Stuart Presnell. Does homotopy type theory provide a foundation for mathematics? *The British Journal for the Philosophy of Science*, 69(2):377–420, 2018.

[51] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, New York, NY, USA, 1986.

[52] M. H. Löb and S. S. Wainer. Hierarchies of number-theoretic functions. I. *Archiv für mathematische Logik und Grundlagenforschung*, 13(1):39–51, Mar 1970.

[53] P. L. Lumsdaine. Weak ω-categories from intensional type theory. *Typed Lambda Calculi and Applications*, pages 172–187, 2009.

[54] P. L. Lumsdaine and M. Shulman. Semantics of higher inductive types. *ArXiv e-prints*, May 2017.

[55] S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer New York, 1994.

[56] A. A. Markov. On constructive mathematics. In *Problems of the constructive direction in mathematics. Part 2. Constructive mathematical analysis*, volume 67 of *Trudy Mat. Inst. Steklov.*, pages 8–14. Acad. Sci. USSR, Moscow–Leningrad, 1962.

[57] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73 – 118. Elsevier, 1975.

[58] Per Martin-Löf. 100 years of zermelo's axiom of choice: What was the problem with it? *Comput. J.*, 49(3):345–350, May 2006.

[59] C. McLarty. *Elementary Categories, Elementary Toposes*. Clarendon Press, 1992.

[60] N. P. Mendler. Quotient types via coequalizers in martin-löf type theory. In G. Huet and G. Plotkin, editors, Informal Proceedings of the First Workshop on Logical Frameworks, Antibes, May 1990, pages 349–360, 1990, May 1990.

[61] Ulf Norell, Nils Anders Danielsson, and Andreas Abel. The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[62] Piergiorgio Odifreddi. *Classical Recursion Theory*. Elsevier, 1989.

[63] I. Orton and A. M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. *ArXiv e-prints*, December 2017.

[64] Erik Palmgren. Proof-relevance of families of setoids and identity in type theory. *Archive for Mathematical Logic*, 51(1):35–47, Feb 2012.

[65] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 221–230, 2001.

[66] Michael Rathjen. The strength of Martin-Löf type theory with a superuniverse. part i. *Archive for Mathematical Logic*, 39(1):1–39, Jan 2000.

[67] Michael Rathjen. The strength of martin-löf type theory with a superuniverse. part ii. *Archive for Mathematical Logic*, 40(3):207–233, Apr 2001.

[68] B. Reus and Th. Streicher. *General synthetic domain theory — A logical approach (extended abstract)*, pages 293–313. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[69] Bernhard Reus. Formalizing synthetic domain theory. *Journal of Automated Reasoning*, 23(3):411–444, Nov 1999.

[70] Emily Riehl and Dominic Verity. Homotopy coherent adjunctions and the formal theory of monads. *Advances in Mathematics*, 286:802 – 888, 2016.

[71] E. Rijke. The join construction. *ArXiv e-prints*, January 2017.

[72] E. Rijke, M. Shulman, and B. Spitters. Modalities in homotopy type theory. *ArXiv e-prints*, June 2017.

[73] Egbert Rijke. Homotopy type theory. Master's thesis, Utrecht University, 2012.

[74] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

[75] G. Rosolini. *Continuity and Effectiveness in Topoi*. PhD thesis, University of Oxford, 1986.

[76] Dana S. Scott. A type-theoretical alternative to iswim, cuch, owhy. *Theor. Comput. Sci.*, 121(1-2):411–440, December 1993.

[77] Robert AG Seely. Locally cartesian closed categories and type theory. In *Mathematical proceedings of the Cambridge philosophical society*, volume 95, pages 33–48. Cambridge University Press, 1984.

[78] Anton Setzer. *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe*. PhD thesis, Ludwig-Maximilians-Universität München, 1993.

[79] Anton Setzer. Well-ordering proofs for Martin-Löf type theory. *Annals of Pure and Applied Logic*, 92(2):113 – 159, 1998.

[80] Anton Setzer. Extending Martin-Löf type theory by one Mahlo-universe. *Archive for Mathematical Logic*, 39(3):155–181, Apr 2000.

[81] Jan M. Smith. The independence of peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic*, 53(3):840–845, 1988.

[82] Thomas Streicher. How intensional is homotopy type theory? In Maria del Mar González, Paul C. Yang, Nicola Gambino, and Joachim Kock, editors, *Extended Abstracts Fall 2013*, pages 105–110, Cham, 2015. Springer International Publishing.

[83] Paul Taylor. The fixed point property in synthetic domain theory. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 152–160. IEEE, 1991.

[84] A. S. Troelstra. A note on non-extensional operations in connection with continuity and recursiveness. *Indag. Math.*, 39(5):455–462, 1977.

[85] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics*. Number v. 1 in Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1988.

[86] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[87] Tarmo Uustalu and Niccolo Veltri. Partiality and container monads. In *Asian Symposium on Programming Languages and Systems*, pages 406–425. Springer, 2017.

[88] Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

[89] Jaap van Oosten and Alex K. Simpson. Axioms and (counter)examples in synthetic domain theory. *Annals of Pure and Applied Logic*, 104(1):233 – 278, 2000.

[90] Vladimir Voevodsky. Resizing rules. Plenary Lecture at TYPES2011, 2011.

[91] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath*: Univalent Mathematics. Available at `https://github.com/UniMath`.

[92] Michael A Warren. The strict ω-groupoid interpretation of type theory. *Models, Logics, and Higher-dimensional Categories: A Tribute to the Work of Mihály Makkai*, 53:291–340, 2011.